UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

GoThings: Uma Arquitetura de *Gateway* de Camada de Aplicação para a Internet das Coisas

Wagner Luís de Araújo Menezes Macêdo

São Cristóvão

2016

UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Wagner Luís de Araújo Menezes Macêdo

GoThings: Uma Arquitetura de *Gateway* de Camada de Aplicação para a Internet das Coisas

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação (PROCC) da Universidade Federal de Sergipe (UFS) como parte de requisito para obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Tarcísio da Rocha

São Cristóvão
2016

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL
UNIVERSIDADE FEDERAL DE SERGIPE

Macêdo, Wagner Luís de Araújo Menezes
A141g GoThings : uma arquitetura de *gateway* de camada de aplicação para a internet das coisas / Wagner Luís de Araújo Menezes Macêdo ; orientador Tarcísio da Rocha. - São Cristóvão, 2016.
70 f. : il.

Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Sergipe, 2016.

1. Rede de computadores - protocolos. 2. Interconexão em redes (Telecomunicações). 3. Protocolo de aplicação sem fio (Protocolo de rede de computador). 4. Internet. 5. Arquitetura de redes de computadores. I. Rocha, Tarcísio da, orient. II. Título.

CDU 004.738.2.057.4

Wagner Luís de Araújo Menezes Macêdo

GoThings: Uma Arquitetura de *Gateway* de Camada de Aplicação para a Internet das Coisas

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação (PROCC) da Universidade Federal de Sergipe (UFS) como parte de requisito para obtenção do título de Mestre em Ciência da Computação.

Trabalho aprovado. São Cristovão, 29 de fevereiro de 2016:

Prof. Dr. Tarcísio da Rocha
Orientador

Prof. Dr. Ricardo José Paiva de Britto Salgueiro
Membro

Prof. Dr. Fábio Luciano Verdi
Membro

São Cristovão
2016

Este trabalho é dedicado àqueles
que amam compartilhar.

Agradecimentos

Foram dois anos de trabalho. Eu devo a muitos por ter alcançado a meta. Primeiramente a Deus, pois sem ele eu nem estaria aqui.

Agradeço à minha querida Patrícia, aquela por quem eu me apaixonei perdidamente, pela paciência, mas também por ter me levantado tantas vezes naqueles momentos mais difíceis.

Às minhas meninas Ana Maria e Maria Alice por me darem mais um motivo para lutar.

Agradeço aos meus pais pelo apoio e suporte durante todo o trajeto. Sem a sua ajuda eu não teria conseguido.

Aos meus irmãos Humberto, Cíntia e Daniela pela compreensão.

Ao meu orientador, Tarcísio, pelo seu incentivo e atenção quando eu mais precisei.

Por último, mas não menos importante, eu agradeço ao PROCC, aos professores, aos colegas de mestrado e a todos aqueles que me ajudaram de alguma forma, em especial ao professor Edward, pelo auxílio no início desse projeto.

Meu muito obrigado!

Resumo

Com a Internet das Coisas (IoT), é previsto que o número de dispositivos conectados atingirá o número de 50 bilhões até 2020. Muitos desses dispositivos adotam, na camada de aplicação, protocolos de mensagem mutualmente incompatíveis entre si. Uma possível solução a esse problema é usar um mesmo protocolo de mensagem em todos os dispositivos. No entanto, um único protocolo nem sempre é adequado para dispositivos restritos e não restritos ao mesmo tempo. Diversas soluções à questão da interoperabilidade na IoT foram propostas, mas elas ou não proveem interoperabilidade transparente ou não são extensíveis e configuráveis o suficiente. Esta dissertação apresenta uma proposta de uma arquitetura para o desenvolvimento de *gateways*, a qual denominamos de GoThings, que permite habilitar a interconectividade entre diferentes protocolos de mensagem. A arquitetura proposta é focada na extensibilidade, configurabilidade e generalidade, no contexto de problemas da IoT.

Palavras-chaves: Dispositivos Restritos, Internet das Coisas, Protocolos de Mensagem, Redes de Computadores.

Abstract

With the Internet of Things (IoT), it is predicted that the number of connected devices will reach 50 billion by 2020. Many of these devices often adopt, at application layer, mutually incompatible messaging protocols. A possible solution to this problem is to use the same messaging protocol among all devices. However, a single protocol is not always suitable for both constrained and unconstrained devices. Several solutions to the interoperability issue in the IoT have been proposed, but they neither provide transparent interoperation nor are extensible and configurable enough. Meanwhile, this paper proposes GoThings, a preliminary gateway architecture which can enable interconnectivity between different messaging protocols. GoThings is focused on extensibility, configurability and generality, in the context of IoT problems.

Key-words: Computer Networks, Constrained Devices, Internet of Things, Message Protocols.

Lista de figuras

Lista de tabelas

Lista de abreviaturas e siglas

API	*Application Programming Interface*
CoAP	*Constrained Application Protocol*
CRUD	*Create Read Update Delete*
ESB	*Enterprise Service Bus*
HTTP	*Hypertext Transfer Protocol*
IETF	*Internet Engineering Task Force*
IoT	*Internet of Things*
JSON	*JavaScript Object Notation*
MIME	*Multipurpose Internet Mail Extensions*
MQTT	*Message Queuing Telemetry Transport*
QoS	*Quality of Service*
REST	*Representational State Transfer*
RSSF	Rede de Sensores sem Fio
SOAP	*Simple Object Access Protocol*
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*
URI	*Uniform Resource Identifier*
WoT	*Web of Things*

Sumário

1 Introdução

Com a Internet das Coisas (IoT[1]), é previsto o número de 50 bilhões de dispositivos conectados até 2020 (EVANS, 2011). A IoT pode ser vista como a visão de um mundo onde todos os objetos físicos (coisas) terão alguma forma de conectividade com a Internet.

Esse número imenso de dispositivos vem acompanhado de uma grande quantidade de padrões concorrentes em um cenário ainda extremamente fragmentado (PRIESTLEY, 2015). Isso significa que diferentes dispositivos podem não ser interoperáveis entre si. Essa heterogeneidade ocorre em todas as camadas de rede, mas especificamente na camada de aplicação já há uma gama de protocolos de mensagem mutualmente incompatíveis (KARAGIANNIS et al., 2015).

Diante disso, o problema da interoperabilidade é ainda maior porque os protocolos podem seguir regras muito divergentes de troca de mensagens, tais como requisição-resposta e publicar-assinar, o que torna muito mais desafiador as soluções de integração (RODRÍGUEZ-DOMÍNGUEZ et al., 2012).

Uma possível solução para esse problema é usar um mesmo protocolo de mensagem em todos os dispositivos. Em algumas abordagens, por exemplo, HTTP (*Hypertext Transfer Protocol*) foi escolhido para integrar todos os dispositivos. Contudo, HTTP nem sempre é adequado a diversos participantes da IoT, como os dispositivos sensores, já que não conseguem trabalhar com qualquer protocolo devido a restrições severas de memória, processamento e energia (DARGIE; POELLABAUER, 2010). O contrário também é verdade, i.e. um protocolo para dispositivos restritos geralmente é inadequado para os não restritos.

Alguns desses dispositivos restritos são limitados pela bateria. Para economizar energia, eles são comumente configurados para dormir com alguma frequência e ficarem ativos somente quando estritamente necessário. Isso impõe alguns obstáculos, já que os dispositivos nem sempre estarão ativos para responder.

Propostas já foram apresentadas para solucionar o problema da interoperabilidade na IoT, mas não pudemos encontrar uma solução que permita uma interoperação transparente bidirecional entre dispositivos restritos com diferentes protocolos de mensagem e que seja extensível para qualquer tipo de protocolo independentemente do padrão de troca de mensagens. Quando falamos transparente, nos referimos a uma solução que provê uma comunicação virtualmente direta, i.e. o emissor não é ciente da presença de um intermediário.

[1] *Internet of Things*, do original em inglês.

1.1 Objetivos

O principal objetivo deste trabalho é propor uma arquitetura para *gateways* capazes de interconectar diferentes protocolos de mensagem para a IoT atendendo aos seguintes requisitos:

- Suporte multiprotocolo por uma estrutura extensível via *plugins*, o que reduz a complexidade de adicionar novos protocolos.
- Suporte aos padrões de troca de mensagens requisição-resposta e publicar-assinar.
- Suporte a configuração através de uma estrutura que permita adicionar ou alterar os comportamentos de um *gateway* dependendo do contexto.
- Suporte a cache para evitar acessos desnecessários a redes restritas, bem como para possibilitar requisições de dados a dispositivos não ativos.

1.1.1 Objetivos Específicos

Para dar suporte ao objetivo geral, os seguintes objetivos específicos precisam ser atendidos:

- Análise comparativa de alguns dos principais protocolos de mensagem na IoT.
- Implementação do *gateway*, como prova de conceito da arquitetura.
- Avaliação dos requisitos da arquitetura.
- Avaliação de desempenho da implementação.

1.2 Justificativa

O cenário visionado para a Internet das Coisas é bastante amplo. Mas a heterogeneidade dos protocolos de mensagem para a IoT já é uma realidade (FOSTER, 2014). Existem diversos protocolos para a IoT, cada um mais adequado para certo perfil de uso. O Capítulo 2 revisa uma seleção de alguns desses protocolos.

Uma interoperabilidade entre diferentes dispositivos é desejada. Sendo assim, não basta que as camadas de transporte, de rede e de enlace sejam interoperáveis. Na camada de aplicação, representado pelos diferentes protocolos de mensagem, também é necessária a atenção. Para resolver esse problema, algumas propostas já foram feitas. O Capítulo 3 analisa alguns desses trabalhos, relacionando-os com a proposta.

Frente aos problemas das soluções existentes para proverem interoperabilidade, no contexto da IoT, justificamos este trabalho com a necessidade de uma solução que resolva essas limitações.

1.3 Organização

Esta dissertação está organizada da seguinte forma:

- Capítulo 2: apresenta os conceitos importantes para a compreensão do tema da dissertação.
- Capítulo 3: discute sobre alguns trabalhos relacionados ao tema da nossa proposta.
- Capítulo 4: apresenta a solução proposta para o problema da interoperabilidade na camada de aplicação.
- Capítulo 5: apresenta a avaliação e discute os resultados.
- Capítulo 6: apresenta as considerações finais e discute possíveis trabalhos futuros.

2 Conceitos Básicos

Este capítulo apresenta o embasamento teórico usado nessa pesquisa. Os conceitos vistos aqui foram fundamentais para o projeto da arquitetura e implementação da solução proposta. Inicialmente, é discutida sobre a definição da Internet das Coisas. Em seguida, são apresentados conceitos da troca de mensagens em rede, finalizando com uma discussão sobre interoperabilidade.

2.1 Internet das Coisas

Devido à grande quantidade de conceitos envolvidos, encontrar a definição exata do que é a Internet das Coisas (IoT) não é fácil, nem se sabe se é possível, opiniões divergem nos textos científicos e técnicos da área. O termo IoT, na verdade, não é tão recente, surgiu em 1999, associado à nova ideia na época de usar etiquetas inteligentes no suporte à logística (SHENG et al., 2013), porém o conceito evoluiu e tornou-se mais abrangente e diversificado. A IoT é também referida por alguns como Internet dos Objetos.

Na visão da Cisco (EVANS, 2011), a IoT é simplesmente o momento no tempo quando haverão mais "coisas (ou objetos)" do que pessoas conectadas à Internet. Usando essa premissa, a própria Cisco indica que a IoT "nasceu" por volta de 2009, quando o número de dispositivos por pessoa chegou a mais de um.

Para Sheng et al. (2013), o termo é um novo sistema de comunicação onde a Internet estará conectada ao mundo físico através das redes de sensores sem fio (RSSF). Os dispositivos sensores podem ser considerados os participantes mais limitados da IoT, pois geralmente são restritos em energia, processamento e memória (DARGIE; POELLABAUER, 2010).

Já conforme Castellani, Fossati e Loreto (2012), a IoT é um conjunto de protocolos e tecnologias que habilitam a Internet a ser usada para conectar pessoas a coisas e coisas a coisas.

Dentre outras, uma das definições mais difundidas entre as pessoas é a de Botterman (2009), em que coloca a IoT como uma rede mundial de objetos interconectados e unicamente endereçáveis, baseados em protocolos de comunicação padronizados.

Uma coisa em comum em quase todos os textos é que concordam que a IoT é o próximo passo da Internet.

2.2 Troca de Mensagens

As aplicações de rede, na maioria das vezes, utilizam para transmitir dados uma abstração chamada mensagem (COULOURIS et al., 2013). Essa abstração se faz necessária para poder oferecer aos programadores uma forma mais fácil de desenvolver suas aplicações, de forma que eles precisem menos ou nenhuma vez operar com as camadas mais baixas de rede.

Toda aplicação de rede utiliza pelo menos um formato de mensagem e regras para trocar mensagens entre os nós de rede. Esse conjunto de regras e formato são chamados de protocolos de aplicação ou também de protocolos de mensagem. Usamos os dois termos neste trabalho intercambiavelmente.

2.2.1 Padrões de Troca de Mensagem

A partir da observação das regras utilizadas em diversos protocolos, foram notadas semelhanças que evoluíram para padrões documentados. Os padrões de troca de mensagem mais utilizados são o Requisição-Resposta (RR) e Publicar-Assinar (PA), brevemente descritos a seguir.

Requisição-Resposta é um paradigma simples largamente usado em sistemas distribuídos (KARAGIANNIS et al., 2015). A ideia desse padrão é que um cliente requisita uma informação a um servidor que então responde com a informação requerida. O popular protocolo HTTP é um representante desse padrão. O foco no padrão RR é na interação fim-a-fim.

Publicar-Assinar emula o procedimento humano de assinar a uma publicação (KARAGIANNIS et al., 2015). A ideia é que do momento em que um assinante expressa interesse em certa informação, ele automaticamente receberá uma cópia da informação a cada vez que ela estiver disponível. O funcionamento é normalmente através de um servidor chamado *broker*, que mantém as assinaturas e recebe as publicações, encaminhando-as a todos os interessados. O foco no padrão PA é no dado.

Uma das diferenças mais relevantes entre RR e PA é o acoplamento do emissor e do receptor. Enquanto no RR sabe-se quem é o receptor, em interações um-para-um, no PA os publicadores e assinantes são completamente desacoplados, em interações um-para-muitos.

2.2.2 Protocolos de Mensagem

Há protocolos de mensagem específicos de uma aplicação e há aqueles que são mais genéricos, prontos para vários cenários de uso. Esta seção apresenta três dos protocolos genéricos mais populares nos trabalhos relativos à IoT: HTTP, CoAP (*Constrained Application Protocol*) e MQTT (*Message Queuing Telemetry Transport*).

2.2.2.1 HTTP

Como provavelmente o protocolo mais popular da Internet, o HTTP (FIELDING; RESCHKE, 2014) adota ao padrão requisição-resposta em um modelo de comunicação síncrono. O HTTP chama os caminhos para acessar um dado de recurso.

Os principais tipos de requisição que um cliente HTTP pode fazer são:

1. `GET`: solicita o conteúdo de um recurso no servidor.
2. `PUT`: solicita a atualização do conteúdo de um recurso no servidor.
3. `POST`: solicita uma ação que é específica do servidor, mas é comumente associada à ação de criar um recurso.
4. `DELETE`: solicita a remoção de um recurso.

Embora o HTTP seja listado como um protocolo para a IoT, o seu foco não é nos dispositivos restritos. Na verdade, no contexto da IoT, é mais comum ver o HTTP como um sistema legado ainda necessário.

2.2.2.2 CoAP

A Internet Engineering Task Force (IETF) projetou o protocolo CoAP (SHELBY; HARTKE; BORMANN, 2014) para satisfazer as demandas por um protocolo leve, mas também interoperável com HTTP, mantendo as suas semânticas praticamente as mesmas. CoAP, assim como HTTP, é um protocolo requisição-resposta e síncrono.

A principal diferença entre os dois é que CoAP é implementado sobre UDP (*User Datagram Protocol*), em vez de TCP (*Transmission Control Protocol*), reduzindo o *overhead* e conseguindo uma comunicação mais leve. Mas CoAP também possui alguns recursos extras importantes para dispositivos restritos, como comunicação *multicast* e a requisição *Observe*, um tipo de requisição equivalente a assinar, onde o servidor pode responder somente quando o recurso for alterado.

Como CoAP executa sobre um transporte não confiável (UDP), um mecanismo para garantir a confiabilidade foi definido. As mensagens podem ser marcadas com o tipo da mensagem, indicando entre outras coisas o nível de QoS (*Quality of Service*). Esses tipos são:

1. Confirmável (CON): uma mensagem que requer o envio da confirmação (ACK). A resposta pode ser enviada de forma síncrona, com a confirmação ou se precisar de tempo para processar, de forma assíncrona em uma mensagem separada.
2. Não Confirmável (NON): uma mensagem que não requer confirmação.
3. Confirmação (ACK): confirma o recebimento de uma mensagem.
4. *Reset* (RST): confirma o recebimento de uma mensagem que não pôde ser processada.

Tabela 2.1 – Características dos protocolos de mensagem.

Protocolo	**Padrão de Troca de Mensagens**	**Modelo de Comunicação**	**Transporte**	**QoS**	**Restrito?**
HTTP	Requisição-Resposta	Síncrono	TCP	–	Não
CoAP	Requisição-Resposta	Síncrono, Resposta assíncrona	UDP	1. Não confirmável 2. Confirmável	Sim
MQTT	Publicar-Assinar	Assíncrono	TCP	1. *Fire and forget* 2. *At least once* 3. *Exactly once*	Sim

Contiki (CONTIKI. . . , 2014), um dos principais sistemas operacionais para a IoT, adota CoAP como seu protocolo de aplicação primário.

2.2.2.3 MQTT

Também importante à IoT é o MQTT (BANKS; GUPTA, 2014), um protocolo publicar-assinar criado pela IBM, projetado para ser simples, leve e adequado a ambientes limitados. A implementação de MQTT é sobre TCP, dificultando a sua adoção em dispositivos extremamente restritos como alguns sensores sem fio, o que levou ao desenvolvimento de MQTT para Redes de Sensores Sem Fio (HUNKELER; TRUONG; STANFORD-CLARK, 2008).

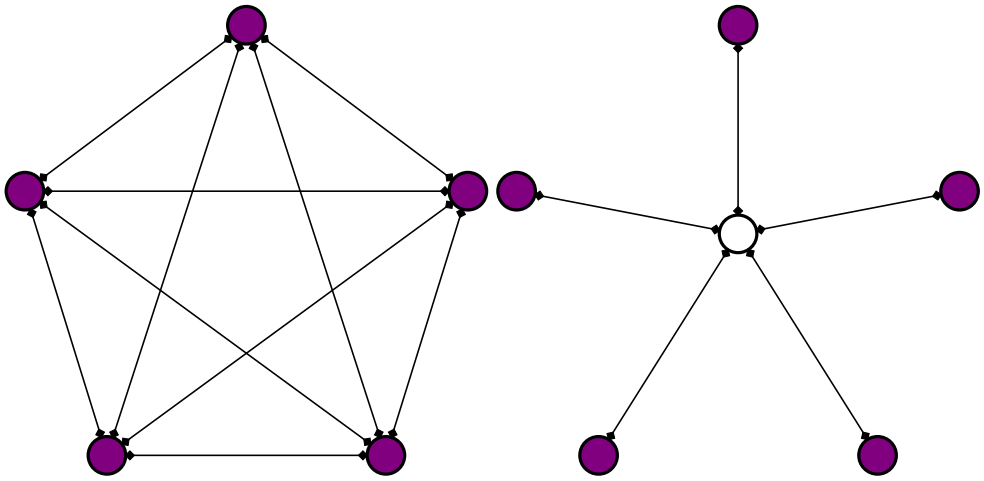
MQTT é um protocolo publicar-assinar assíncrono, baseado em um *broker* que contém tópicos. Os tópicos são equivalentes ao caminho do recurso no HTTP, mas os clientes podem publicar e assinar.

Esse protocolo garante a confiabilidade provendo três níveis de QoS:

1. *Fire and forget*: a mensagem é enviada, mas nenhuma confirmação de que ela foi entregue aos assinantes é requerida.
2. *Delivered at least once*: a mensagem é enviada pelo menos uma vez aos assinantes.
3. *Delivered exactly once*: é garantida que os assinantes irão receber somente uma mensagem. Este é um mecanismo complexo e utiliza mais banda, pois duas confirmações são requeridas.

2.2.2.4 Comentário

Como cada protocolo tem as suas próprias características, um mapeamento entre os mesmos é um desafio necessário para a proposta do *gateway* deste trabalho. No Capítulo 4 a nossa proposta é apresentada e pode ser visto como os mapeamos. A Tabela 2.1 exibe as características de cada protocolo apresentado.



(a) Pontes diretas. (b) Pontes indiretas.

Figura 2.1 – Abordagens para prover interoperabilidade.

2.2.3 Troca de Mensagens na IoT

Nesse trabalho, consideramos que a maioria das aplicações da IoT exigem pouca mobilidade e mensagem pequenas, na ordem de poucos *bytes* a alguns *kilobytes*, o que é confirmado pelos resultados em Elmangoush et al. (2015). Assim, assumimos como o padrão de interação da arquitetura proposta esses requisitos modestos.

2.3 Interoperabilidade

O termo interoperabilidade refere-se à habilidade para um ou mais sistemas ou domínios de conectar, entender e trocar dados com outro para um dado propósito (BLAIR et al., 2011). O objetivo em prover interoperabilidade entre diferentes protocolos é permitir a comunicação entre esses diferentes domínios mantendo a mensagem transferida inalterada ou sem perder a sua semântica original.

Nesse caso, para haver interoperabilidade, é requerido mapear os diferentes comportamentos de cada domínio. Esse mapeamento é usualmente chamado de ponte. Na literatura, as abordagens para interoperabilidade entre diferentes domínios reduzem-se basicamente a duas: pontes diretas e pontes indiretas (ISSARNY; BENNACEUR; BROMBERG, 2011).

2.3.1 Pontes Diretas

As pontes diretas (veja a Figura 2.1a) consistem em uma tradução direta dos comportamentos entre os domínios, assim um tradutor dedicado é requerido para cada par de domínios envolvidos. A vantagem de uma ponte direta é requerer somente uma operação de tradução, mas o número necessário de tradutores cresce de forma quadrática ao número n de domínios, como indica a equação 1.

$$t(n) = \frac{n(n-1)}{2} \tag{1}$$

2.3.2 Pontes Indiretas

Em contraste, nas pontes indiretas um formato intermediário é usado. Dessa forma, os comportamentos de cada domínio são mapeados por somente um tradutor para um domínio intermediário (veja a Figura 2.1b). Embora essa abordagem tenha a desvantagem de requerer duas operações de tradução, já que toda mensagem é primeiramente traduzida para o formato intermediário antes de ser traduzida para o formato do domínio destino, o número de tradutores necessários é linear ao número de domínios, o que de fato é a grande vantagem em usar pontes indiretas.

Neste trabalho, adotamos a abordagem de ponte indireta. A razão dessa decisão é explicada na Seção 4.2.2.

3 Trabalhos Relacionados

Diversas abordagens foram propostas para solucionar o problema da interoperabilidade de protocolos de mensagem na IoT. Este capítulo apresenta os trabalhos mais relevantes à nossa proposta, realçando os pontos positivos e falhas, dando destaque a como nossa proposta pode solucioná-las.

3.1 Modelo de Comunicação Híbrido para Sistemas Ubíquos

Rodríguez-Domínguez et al. (2012) propõem um modelo de comunicação para integrar os paradigmas requisição-resposta e publicar-assinar, no intuito da implementação de um *broker* híbrido que abstrai outros *brokers* mais específicos.

Baseado num estudo realizado pelos autores, nem requisição-resposta nem publicar-assinar são suficientes para preencher todos os requisitos usualmente requeridos por sistemas ubíquos. Eles concluem que a maioria desses sistemas na verdade precisam de uma combinação dos dois.

O modelo proposto contribui em aproveitar o melhor de cada um dos padrões de troca de mensagem. O resultado é utilizado na confecção de um *middleware* utilizado no *broker* que é capaz de escolher o paradigma mais adequado, dependendo da necessidade de comunicação. Por exemplo, requisição-resposta é utilizado, quando apenas um destinatário é escolhido.

As maiores diferença desse trabalho para a nossa proposta é na utilização de um *middleware* e a abstração é para a comunicação entre outros *middlewares*. A nossa proposta, por outro lado, tem como foco os protocolos de mensagem.

3.2 *Web* das Coisas

Conforme Guinard, Trifa e Wilde (2010), a *Web* das Coisas, ou simplesmente WoT (*Web of Things*), é um refinamento da Internet das Coisas onde a preocupação não é só pôr os dispositivos em rede, mas também integrá-los a um mesmo protocolo.

A WoT coloca a *web* como um paradigma comum entre os dispositivos. Os autores consideram que tecnologias *web*, tais como REST[1] (FIELDING; TAYLOR, 2002), são as mais efetivas para uma verdadeira IoT, isto é, onde todas as coisas realmente interagem entre si. Nessa proposta, quando dispositivos não podem executar um servidor HTTP, a integração é feita usando intermediários chamados *Smart Gateways*, que são pontes de *software* que

[1] *Representational State Transfer*

abstraem a comunicação heterogênea entre diferentes protocolos através de uma API (*Application Programming Interface*) baseada em REST.

A abordagem da WoT também suporta interações publicar-assinar via *feeds web*. A nossa proposta é diretamente relacionada com o *Smart Gateway* da WoT, mas é mais genérica, pois não é limitada a uma interface REST.

3.3 *Proxy* HTTP-CoAP

Castellani, Fossati e Loreto (2012) apresentam diretrizes para o mapeamento entre os protocolos HTTP e CoAP, no tocante à implementação de um *proxy* cruzado entre os dois protocolos. A principal motivação do trabalho é suplementar a *Web* das Coisas com o CoAP, muito mais leve que o HTTP, de modo a criar uma ponte entre as redes convencionais e as limitadas.

Esse tipo de *proxy* pode interconectar protocolos de forma transparente e, principalmente, mantendo quase toda a semântica, mas é limitado a protocolos com interfaces parecidas, que é o caso dos protocolos HTTP e CoAP.

3.4 *Broker* MQTT-HTTP

Collina, Corazza e Vanelli-Coralli (2012) consideram que os requisitos para tornar a IoT uma realidade não podem ser atingidos utilizando um único protocolo de comunicação.

Para tanto, os autores propuseram um novo tipo de *broker* publicar-assinar baseado em tópicos onde o acesso ao conteúdo dos tópicos pode ser feito por múltiplos protocolos. A proposta foi validada com a implementação de um *broker* que responde aos protocolos HTTP e MQTT.

A comunicação é baseada no padrão publicar-assinar, de forma que a solução é ótima para a disseminação de dados. Porém, apesar dessa proposta prover interoperabilidade entre os protocolos, uma interação direta entre os dispositivos é bloqueada, já que a comunicação é voltada às interações através dos conteúdos do *broker*. Isto quer dizer que a interligação entre dispositivos não existe, ou seja, um dispositivo não pode requisitar algo de outro dispositivo.

3.5 Gerador de *Gateways z2z*

Bromberg et al. (2009) propuseram *z2z*, uma sofisticada abordagem geradora para construir *gateways* genéricos, que é composta por uma linguagem específica, um sistema de tempo de execução e um compilador. O objetivo é reduzir o trabalho do programador de *gateways*, escondendo problemas complexos de programação de redes, tais como respostas assíncronas.

Para usar o *z2z*, o programador utiliza a linguagem provida para descrever como cada protocolo interage com a rede, como as mensagens dos protocolos são estruturadas e como traduzir as mensagens entre cada protocolo. Os *gateways* são estaticamente gerados para código C a partir dessas especificações.

A proposta usa a abordagem de pontes diretas para interoperação dos protocolos. Os autores afirmam que pontes indiretas também são suportadas, porém, na prática, isso é feito através de dois *gateways* interligados.

Assim, embora *z2z* ofereça uma abordagem extremamente customizada, os *gateways* gerados apenas interligam dois protocolos. Além disso, requisição-resposta parece ser o único padrão de troca de mensagem suportado.

3.6 Sobre *Middleware* e *Enterprise Service Bus* (ESB)

Esta seção discute brevemente duas outras abordagens gerais que também podem ser consideradas alternativas à interoperabilidade entre protocolos heterogêneos: *middleware* e ESB.

Um *middleware* é uma abstração de programação que mascara a heterogeneidade das redes, do *hardware* e dos sistemas operacionais (COULOURIS et al., 2013). Para isto, o *middleware* oferece um modelo de computação para ser usado pelos programadores, ou seja, para que dois nós de rede se comuniquem, é preciso que estejam utilizando o mesmo *middleware*.

O objetivo principal entre a nossa solução e o *middleware* é o mesmo: habilitar a interoperabilidade. No entanto, não queremos impor um padrão para ser usado nos dispositivos. Pelo contrário, queremos dar a liberdade de usar o protocolo mais adequado, sem perder a capacidade de se comunicar com dispositivos que usem protocolos diferentes.

Um ESB (CHAPPELL, 2004; FLURRY; CLARK, 2011) é um barramento interno para permitir que as aplicações e serviços se comuniquem entre si. A ideia é que todas as aplicações se liguem ao barramento para poderem receber as mensagens que os interessam quando publicados por outra aplicação. As aplicações não são responsáveis por se conectarem diretamente entre si, elas publicam mensagens ao barramento e todas as aplicações interessadas as recebem, num estilo de interação publicar-assinar.

A principal semelhança com a nossa proposta é que, normalmente, um ESB permite a ligação com ele por protocolos diferentes, tais como SOAP[2] (GUDGIN et al., 2007) e REST (FIELDING; TAYLOR, 2002), porém eles são tratados meramente como meio de transporte das mensagens no formato do barramento.

O diferencial da nossa proposta em relação ao ESB, é que nesse último os dispositivos precisam se conectar previamente ao barramento para receber mensagens. O nosso *gateway*, por outro lado, promove a interligação entre dispositivos com protocolos heterogêneos sem precisar

[2] *Simple Object Access Protocol*

que os dois lados se conectem previamente ao *gateway*. Basta que um deles requisite ao *gateway*, que a comunicação com o outro lado será intermediada.

3.7 Discussão

Adicionando aos breves comentários de cada seção, fizemos um comparativo de cada uma das soluções e a nossa proposta. Os resultados da comparação pode ser vista na Tabela 3.1. A nossa proposta, GoThings, foi colocada na última linha. O *middleware* não foi adicionado porque quase todos os resultados dependeriam de qual tipo e implementação de *middleware* estaria sendo avaliado. Esses resultados são comentados a seguir.

Quanto aos padrões de troca de mensagem suportados, todas as soluções suportam requisição-resposta, mas nem todas suportam o publicar-assinar. O *Smart Gateway* WoT suporta o último apenas através de *feeds web*. Na nossa proposta é suportado ambos os padrões, sem restrição da tecnologia.

Avaliamos quais protocolos são suportados em cada solução e se é possível estender para mais protocolos. Nota-se a relação entre a capacidade de extensão e a abordagem de interoperabilidade: todas as soluções com essa capacidade adotam a ponte indireta.

A interligação fim-a-fim é o recurso que indica que a solução permite que nós clientes e nós servidores interajam de forma espontânea, i.e. sem que estejam cientes de que estão se comunicando através de um intermediário. O *Smart Gateway* WoT suporta a interligação de forma unidirecional: o foco é apenas para que clientes HTTP acessem sensores que se comunicam por Bluetooth ou Zigbee, mas não permitem que os sensores acessem recursos *web*. As soluções que não são capazes dessa interligação servem apenas para oferecer os dados por mais de um protocolo.

Frente aos problemas das soluções existentes no contexto de *gateways* orientados à IoT, o próximo capítulo apresenta nossa proposta, que espera-se resolver essas limitações.

Tabela 3.1 – Comparação dos recursos entre os trabalhos relacionados e a nossa proposta.

Solução	Requisição-Resposta	Publicar-Assinar	Abordagem	Protocolos	Permite Extensão?	Interligação fim-a-fim?
Modelo de Comunicação Híbrido para Sistemas Ubíquos	Sim	Sim	Ponte Indireta	(interopera *middlewares*)	Sim	Não
***Smart Gateway* WoT**	Sim	Sim, por *feeds web*.	Ponte Indireta	– *Bluetooth* – *Zigbee*	Sim	Sim, unidirecional.
***Proxy* HTTPCoAP**	Sim	Não	Ponte Direta	– HTTP – CoAP	Não	Sim
Broker MQTTHTTP	Sim	Sim	Ponte Indireta	– HTTP – MQTT	Sim	Não
Gerador de *Gateways z2z*	Sim	Não	Ponte Direta	Deve ser especificado	Não	Sim
Enterprise Service Bus	Sim	Sim	Ponte Indireta	Depende do ESB	Sim	Não
GoThings	Sim	Sim	Ponte Indireta	– HTTP – CoAP – MQTT	Sim	Sim

4 Arquitetura GoThings

Neste capítulo é apresentada a proposta desta dissertação. Inicialmente discutimos sobre o que é um *gateway* de camada de aplicação no ponto de vista deste trabalho. A Seção 4.2 dá uma visão geral da arquitetura proposta. A Seção 4.3 descreve o formato de mensagem interno usado no *gateway*. As seções 4.4 a 4.6 abordam o funcionamento interno do *gateway* e a relação com clientes e servidores externos. Por fim, a Seção 4.7 apresenta um protótipo experimental da arquitetura proposta seguido da Seção 4.8 que conclui o capítulo.

4.1 *Gateway* de Aplicação

O objetivo deste trabalho é permitir o desenvolvimento de *gateways* de camada de aplicação para a IoT. Logo, esta sessão descreve em alto nível como deve funcionar tais *gateways*, comparando, para melhor entendimento, com o *gateway* que opera na camada de rede, com base no padrão TCP/IP.

O *gateway* de rede utiliza o IP como meio de interligação, permitindo aos nós de rede ignorarem quais protocolos estão operando na camada de enlace e física (TANENBAUM; WETHERALL, 2011). Da mesma maneira um *gateway* de aplicação tem a função de interligar dois pontos que não poderiam se comunicar diretamente. No entanto, como a camada de aplicação foi projetada para ser flexível, não há uma camada acima que permitiria a interligação dos protocolos de aplicação.

Em consequência disso, a camada que provê a interligação entre os protocolos de mensagem no *gateway* de aplicação deve ser interna. Os *hosts*, portanto, não precisam ter conhecimento dessa camada e podem continuar operando com os seus próprios protocolos. Definimos essa camada e a apresentamos na Seção 4.3.

A arquitetura GoThings permite construir um *gateway* capaz de conversar com todos os protocolos suportados. A Figura 4.1 ilustra um *gateway* capaz de receber e enviar requisições por quatro protocolos de aplicação diferentes. Em alto nível, a figura está mostrando o fluxo de uma requisição bem sucedida de um cliente HTTP para um servidor MQTT, onde os seguintes passos são observados:

Passo 1: O fluxo inicia quando um cliente HTTP requisita ao *gateway* através do módulo HTTP.

Passo 2: A mensagem é encaminhada ao núcleo do *gateway*, que representa a camada de compatibilidade.

Passo 3: O núcleo do *gateway* encaminha a mensagem ao módulo MQTT, após identificar o protocolo de destino.

Passo 4: O módulo MQTT faz uma requisição ao servidor MQTT.

Passo 5: Uma resposta do servidor MQTT contatado é recebida pelo módulo MQTT.

Passo 6: A resposta é encaminhada ao núcleo do *gateway*.

Passo 7: O *gateway* identifica o módulo que originou a requisição e encaminha a resposta para o módulo HTTP.

Passo 8: O módulo HTTP entrega a resposta ao cliente HTTP que originou a requisição.

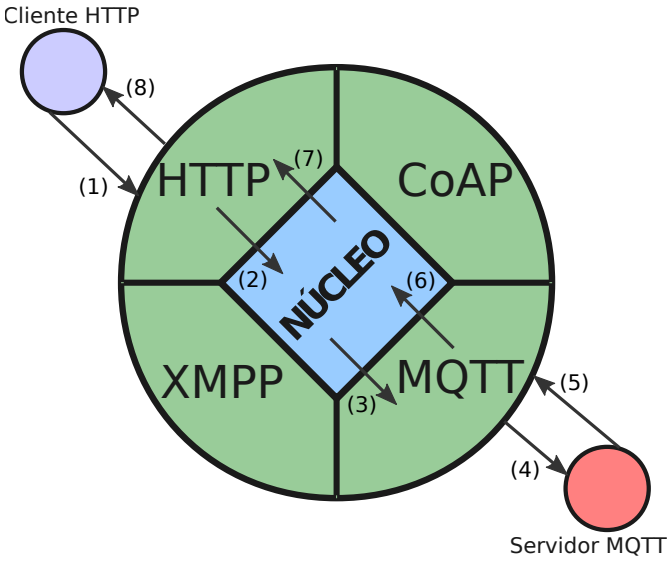


Figura 4.1 – O fluxo atravessando o *gateway* em uma requisição HTTP-MQTT.

4.2 A Arquitetura do *Gateway*

A arquitetura GoThings possibilita a construção de *gateways* tais como o abordado na seção anterior. Esta seção descreve a arquitetura, estendendo a proposta preliminar apresentada em Macêdo, Rocha e Moreno (2015).

4.2.1 Requisitos

Alguns requisitos foram definidos para guiar na definição da arquitetura proposta. Eles podem ser vistos a seguir.

Interoperabilidade. Cada protocolo usa uma linguagem em particular. Isso impossibilita os dispositivos heterogêneos a interoperar, a não ser que cada protocolo necessário à comunicação esteja instalado.

Adicionalmente, é sabido que a complexidade de manter aplicações multiprotocolo é maior que com apenas um protocolo.

De modo que, a arquitetura deve tornar possível desenvolver um *gateway* que faça o intermédio da comunicação entre dispositivos com diferentes protocolos de aplicação.

Extensibilidade. A arquitetura deve permitir a adição de novos protocolos a um *gateway* sem modificação de suas estruturas internas. Assim, deverá ser fácil interconectar protocolos não conhecidos previamente.

Generalidade. A arquitetura deve ser genérica suficiente para permitir seu uso em vários cenários no contexto da IoT.

Configurabilidade. A arquitetura deve prover mecanismos para modificar o comportamento de um *gateway* de forma granular.

Adequação a Dispositivos Restritos. Os dispositivos restritos possuem diversas limitações como memória e processamento, de forma que, geralmente, não conseguem operar com vários protocolos ao mesmo tempo. Além disso, como eles também são restritos em energia, o que impedem de estar ativos o tempo todo, nem sempre estarão disponíveis para responder.

Assim, a arquitetura deve definir recursos para se tornar amigável à comunicação com esses dispositivos.

4.2.2 Abordagem de Interoperabilidade

Justificado pela necessidade de um *gateway* facilmente extensível em termos de protocolos, a arquitetura GoThings segue a abordagem de ponte indireta. Pontes indiretas tornam necessário um formato intermediário como dito na Seção 2.3. Para isso foi definido como formato intermediário um modelo de mensagem com campos e opções comuns de uma interseção de protocolos da IoT e padrões de troca de mensagem. O modelo de mensagem é abordado na Seção 4.3.

4.2.3 Componentes da Arquitetura

A visão geral da arquitetura GoThings pode ser vista na Figura 4.2. Seus componentes principais são brevemente abordados a seguir.

4.2.3.1 *Plugins*

Um *plugin* é um componente de software que implementa algumas interfaces para interagir com o gerenciador de comunicação.

O papel dos *plugins* na arquitetura é de habilitar a interação com o *gateway* por diferentes protocolos de mensagem. Nesse contexto, isso significa que cada *plugin* deve ter:

- um servidor de um dado protocolo ouvindo em alguma porta, aceitando requisições e as respondendo;

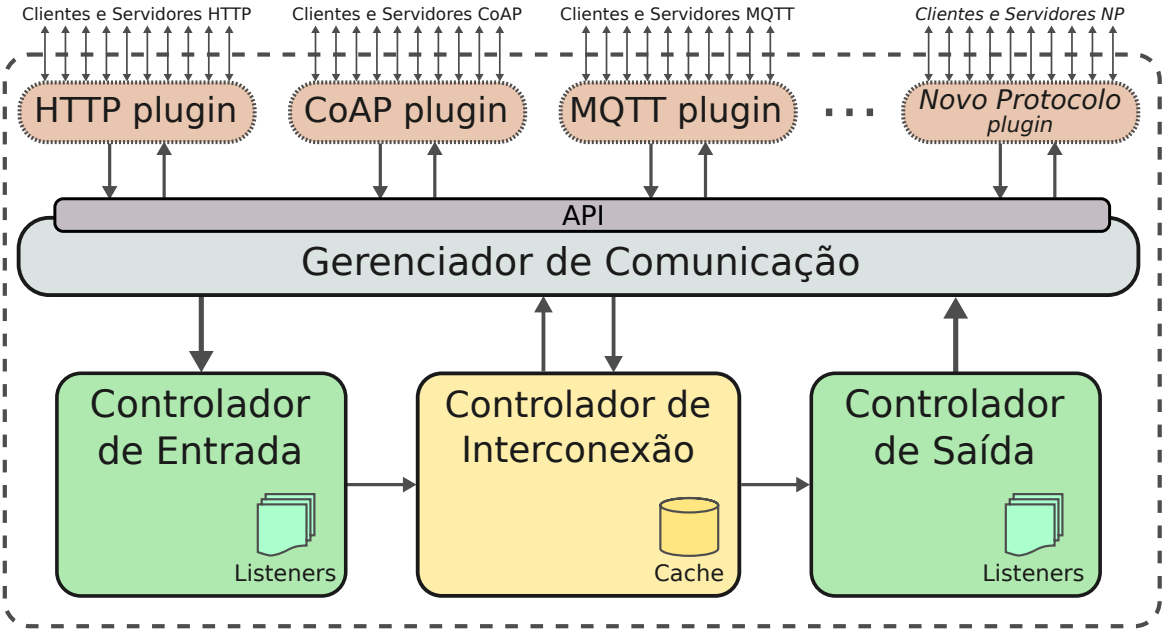


Figura 4.2 – Visão geral da arquitetura GoThings.

- um cliente do mesmo protocolo, pronto para fazer requisições a servidores externos e esperar por respostas.

O subsistema de *plugins* é detalhado na Seção 4.4. Ele foi projetado para incentivar o uso de *softwares* externos, pois como a maioria dos protocolos possuem bibliotecas prontas para uso, isso dá ao programador a oportunidade de acelerar o desenvolvimento de *plugins*.

4.2.3.2 Gerenciador de Comunicação

Este é o componente que faz o intermédio da comunicação entre os *plugins* e os controladores. Ele permite a extensibilidade do *gateway* e abstrai o funcionamento de cada protocolo através de um modelo de mensagem genérico. Note que os *plugins* não podem se comunicar diretamente entre si, mas somente através do gerenciador de comunicação.

4.2.3.3 Controlador de Interconexão

O gerenciador de comunicação é responsável por encaminhar as requisições e devolver as respostas. As diretrizes da comunicação são realmente feitas pelo controlador de interconexão. É responsabilidade desse controlador:

1. identificar o *plugin* de destino de cada requisição;
2. identificar o *plugin* de origem de cada resposta;
3. manter o estado de cada requisição até receber uma resposta;
4. verificar se uma resposta está em cache, e entregá-la se aplicável.

O modo como deve ser identificado o *plugin* de destino é abordado na Seção 4.5.

Cada requisição que chega ao *gateway* recebe um identificador para saber para qual *plugin* devolver a resposta (veja a Seção 4.3.3). A manutenção do estado das requisições é feito pelo controlador através desse identificador. Isso envolve algo como uma estrutura de dados chave-valor em que a chave é o identificador e o valor o nome do *plugin* que originou a requisição.

O cache foi definido como um elemento opcional, no entanto é um importante recurso do componente. A presença de um cache vai garantir menos requisições externas sendo feitas. Isso não quer dizer que menos requisições serão feitas ao *gateway*, já que isso depende de cada protocolo, mas que pode garantir menos requisições sendo originadas do *gateway*.

Em situações onde o *gateway* encontra-se próximo, na mesma rede por exemplo, e o pedido seja para um dispositivo na Internet, o cache interno também beneficiará protocolos que não suportam cache, que deverão receber respostas mais rápidas.

Outro fator determinante é que dispositivos restritos nem sempre estão ativos. Geralmente, em prol da economia de energia, esses dispositivos entram em suspensão de tempos em tempos. Nesses momentos, não há como obter valores deles. Porém, a presença de um cache permitiria o acesso ao último conteúdo.

4.2.3.4 Controladores de Entrada e Saída

Estes são os componentes que fazem um pré-processamento nas requisições e nas respostas que, respectivamente, entram e saem do *gateway*.

Como elementos-chave, os *listeners* são ações disparadas por eventos nos controladores de entrada e saída. Um *listener* pode ser uma simples ação de *log* ou uma ação complexa que pode modificar a mensagem. Na verdade, deverá ser possível até mesmo priorizar ou descartar a mensagem. Um exemplo de um *listener* é dado na Seção 4.5.1.

A arquitetura não define nenhum *listener*, nem quais os eventos disponíveis, isso fica a cargo da implementação. A princípio, todas as mensagens que chegam, passam por todos os *listeners* registrados. Espera-se que a implementação facilite o registro de *listeners* sem precisar modificar diretamente o código do núcleo.

4.2.4 O Trajeto de Uma Requisição

Quando um cliente faz uma requisição a um *gateway*, faz através de um *plugin*. Mas até a resposta ser devolvida ao cliente, algum esforço é necessário. Um esboço desse esforço pode ser visto na Figura 4.3 que mostra um diagrama de colaboração para uma requisição HTTP-MQTT. Comentamos a sequência de passos a seguir:

1: O cliente faz uma requisição HTTP a um *gateway* exatamente como requisitaria a um servidor HTTP comum.

2–3: O *plugin* HTTP recebe a requisição e envia-a ao Gerenciador de Comunicação (GC) que por sua vez encaminha ao Controlador de Entrada (CE).

4: CE envia a mensagem de requisição a todos os *listeners* de entrada registrados e então encaminha ao Controlador de Interconexão (CIC).

5–8: Esta sequência é executada somente se o CIC não encontrar o recurso atual no cache (*cache hit*). Do contrário, se nada foi encontrado (*cache miss*), uma interação com um servidor MQTT é necessário. Isso é feito após o CIC passar a requisição ao GC que encaminha-a ao *plugin* de destino (MQTT neste caso). O *plugin* faz uma requisição ao servidor, e quando resposta chegar o *plugin* repassa-a ao GC que encaminha-a ao CIC que, finalmente, armazena em cache.

9: Seja a resposta recuperada do cache ou não, ela é passada ao Controlador de Saída (CS).

10: Similarmente ao CE, o CS envia a mensagem de resposta a todos os *listeners* de saída registrados e então encaminha ao GC.

11–12: GC envia a resposta ao *plugin* do protocolo de origem (HTTP neste caso) que finaliza a requisição enviando a resposta ao cliente.

4.3 O Modelo de Mensagem

A comunicação que ocorre entre os *plugins* e o *gateway* segue algumas regras específicas em um formato específico de mensagem. O modelo de mensagem descreve esse formato interno, mas também os relacionamentos entre os tipos de mensagem e as regras de interação. Isso faz do modelo descrito nessa seção um dos principais elementos da arquitetura.

4.3.1 Composição da Mensagem

Para obter um modelo o mais simples possível, a mensagem possui apenas dois elementos: *cabeçalhos* e *carga útil*.

Cabeçalhos são os metadados da mensagem, eles são detalhados na Seção 4.3.3.
Carga útil (*payload*) é o conteúdo da mensagem, representado por um vetor de bytes.

4.3.2 Tipos de Mensagem

Para prover uma interação entre nós de rede habilitado para os padrões de troca requisição-resposta e publicar-assinar, mas sem comprometer a simplicidade da solução, optamos por um modelo de mensagem baseado em apenas dois tipos de mensagem: *Requisição* e *Resposta*. Os nomes das mensagens foram baseados fortemente no padrão requisição-resposta, pois é

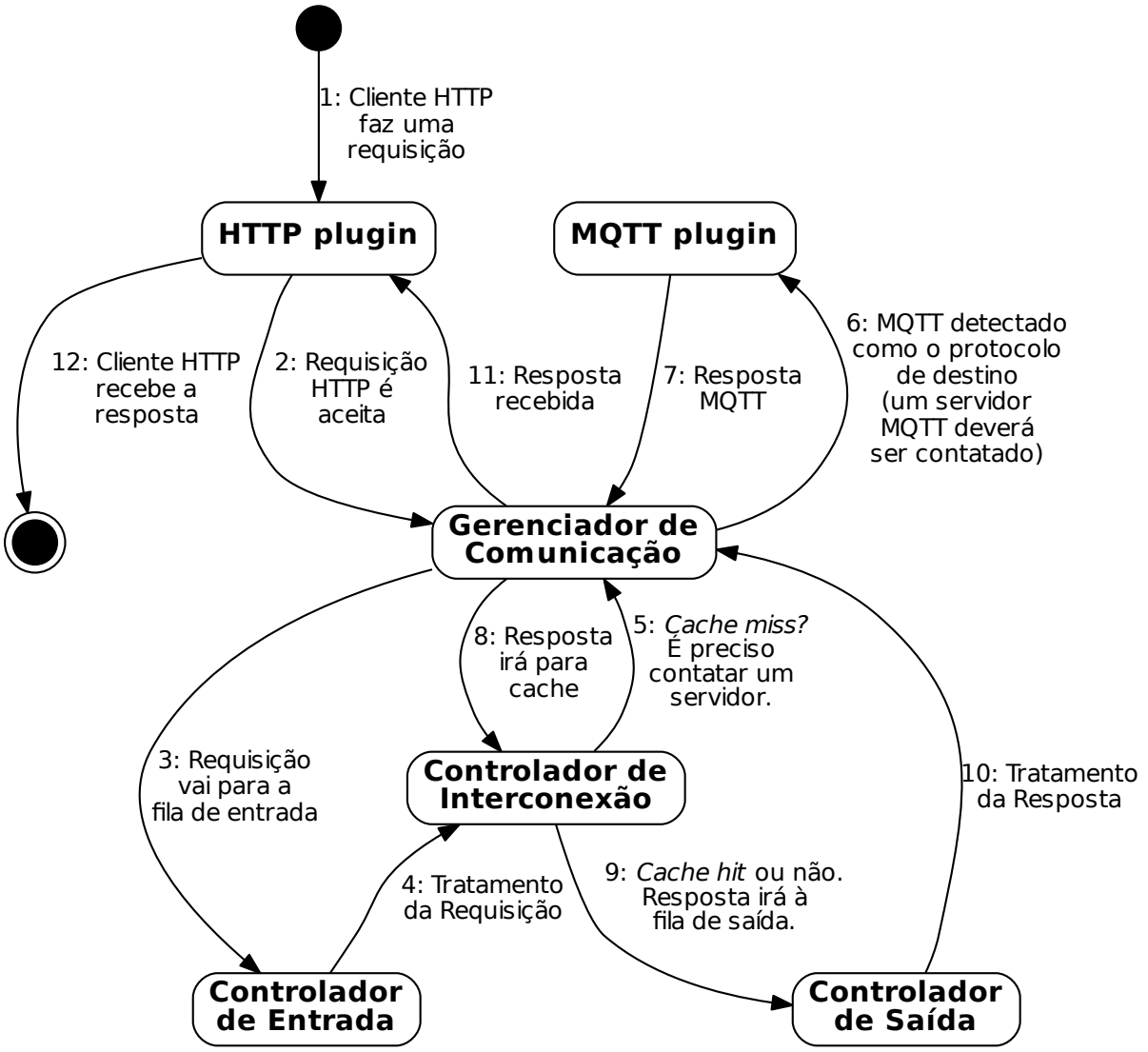


Figura 4.3 – Diagrama de colaboração para uma requisição HTTP-MQTT.

normalmente mais intuitivo que o publicar-assinar. Apesar disso ambos são completamente suportados pelo modelo. Cada tipo de mensagem é explicado a seguir.

Requisição é utilizada para o cliente solicitar uma operação a um recurso em um servidor. Uma requisição pode possuir *payload* ou não, dependendo da operação utilizada.

Resposta é a mensagem de retorno a uma requisição. A operação requisitada indica, além da presença de um *payload*, o número de mensagens de resposta retornadas, conforme é explicado na próxima seção.

Quando uma requisição falha, a mensagem de resposta referencia os cabeçalhos da requisição que falhou adicionado de um código de erro. Nesses casos, o *payload* é sempre ignorado.

Tabela 4.1 – Operações definidas para a mensagem de requisição.

Valor	Operação	Descrição
0	`CRIAR`	Requisita a criação de um conteúdo no destino.
1	`LER`	Requisita o valor atual de um recurso.
2	`ATUALIZAR`	Requisita a atualização do conteúdo de um recurso.
3	`REMOVER`	Requisita a remoção de um recurso.
4	`OBSERVAR`	Requisita o valor atual de um recurso, assim como na operação `LER`, mas monitora por mudanças. Uma nova resposta deve ser recebida a cada alteração.
5	`DESOBSERVAR`	Cancela o monitoramento de um recurso

4.3.3 Cabeçalhos

As mensagens carregam, além de um conteúdo qualquer em um vetor de *bytes*, cabeçalhos que dão significado à mensagem. Os cabeçalhos são um dos elementos mais importantes do modelo de mensagem, pois são através deles que se faz a interseção entre as características dos diferentes protocolos.

Alguns cabeçalhos são obrigatórios, como o que indica a operação da requisição, e serão sempre utilizados por todos os protocolos. Os cabeçalhos opcionais são usados a depender do protocolo e da implementação. Cada um dos cabeçalhos é apresentado a seguir com a descrição, a obrigatoriedade e o tipo de dados utilizado. Esse último é importante para permitir implementações do modelo compatíveis, independente da linguagem de programação.

Sequência (*obrigatório*): é um inteiro de 64 bits único que serve para identificar cada mensagem que entra no *gateway*.

Cada nova mensagem de requisição deve ter um número de sequência gerado por algum mecanismo de controle dentro do *gateway* que garanta números únicos, mas não necessariamente de forma incremental. A função da sequência é de casar as respostas recebidas com as requisições de modo a encaminhá-las corretamente para o cliente que fez a requisição.

Operação (*obrigatório*): este é o cabeçalho mais importante da mensagem, pois o *gateway* deve ter um comportamento diferente dependendo do valor da operação. É equivalente ao "método" do HTTP, mas diferente deste, a operação é um inteiro de 8 bits não sinalizado. Este é um cabeçalho exclusivo da mensagem de requisição, de forma que é ignorado nas respostas.

O principal conjunto de operações foi modelado baseando-se nas operações tradicionais de banco de dados, conhecidas pela sigla CRUD (*Create Read Update Delete*), servindo integralmente à comunicação no padrão requisição-resposta. Essas operações são `CRIAR`,

`LER`, `ATUALIZAR` e `REMOVER`. Adicionalmente foram definidas as operações `OBSERVAR` e `DESOBSERVAR`, baseadas no recurso de mesmo nome do protocolo CoAP, e equivalentes às operações de assinar e cancelar assinatura do padrão publicar-assinar. A ação de publicar pode ser traduzida das operações `CRIAR` ou `ATUALIZAR`. A Tabela 4.1 descreve cada uma das operações e o valor inteiro de representação.

A operação indica a presença ou ausência de *payload* nas mensagens de ida e volta. As únicas mensagens que contém *payload* são as requisições de operações `CRIAR` ou `ATUALIZAR` e as respostas às requisições `LER` ou `OBSERVAR`.

Caminho (*obrigatório*): uma *string* que indica o recurso do alvo requisitado. É um cabeçalho importante por ser usado na identificação do protocolo de destino, conforme discutido na Seção 4.5. É usado na requisição por motivos óbvios, mas também nas respostas para indicar o caminho que foi requisitado.

Alvo (*obrigatório*): uma *string* que indica o IP ou o nome do destino a ser contatado. Este campo só é necessário na mensagem de requisição, porém não deve ser preenchido pelos *plugins*, pois o *gateway* o preenche antes de encaminhar a requisição ao *plugin* de destino, conforme é evidenciado na Seção 4.6.

Tipo de conteúdo (*opcional*): uma *string* que indica, no formato MIME[1] (FREED; BORENSTEIN, 1996), o tipo de conteúdo da mensagem. Esse campo só tem sentido em requisições cujas operações `CRIAR` ou `ATUALIZAR` e nas respostas que possuem *payload*.

Esse campo foi inspirado no cabeçalho *Content-Type* do HTTP. Vários protocolos tais como MQTT não utilizam esse campo.

Tipos esperados (*opcional*): campo inspirado pelo cabeçalho *Accept* do HTTP, possuindo a mesma utilidade, que é de indicar qual o tipo de conteúdo que uma dada requisição espera na resposta. Assim como o tipo de conteúdo, usa o formato MIME. Um *gateway* pode usar esse campo para realizar conversão do tipo de conteúdo antes de entregar a resposta. Obviamente, só é usado nas mensagens de requisição.

QoS (*opcional*): indica o nível de QoS de uma dada requisição. Os níveis de QoS dependem de cada protocolo. Por exemplo, não há opções de QoS no HTTP. O QoS não muda como a resposta é dada, mas como a requisição será tratada pelo protocolo de destino. As respostas não usam esse campo.

Esse campo é um inteiro de 8 bits não sinalizado, iniciando de 0 (indicando QoS desativado) até um número dependente do protocolo. Por exemplo, CoAP tem QoS 0 e 1 (dois níveis) e MQTT tem QoS 0, 1 e 2 (três níveis).

Em uma possível requisição MQTT-CoAP usando QoS 2, o nível 1 seria assumido já que CoAP tem QoS máximo de 1.

[1] *Multipurpose Internet Mail Extensions*

Tabela 4.2 – Códigos de erro.

Código	Nome	Descrição
0	URI inválida	Indica que a *string* do caminho entregue pelo *plugin* não pôde ser lida por algum erro gramatical.
1	*Plugin* indisponível	Indica que não há um *plugin* registrado para o protocolo selecionado.
2	Alvo não encontrado	Indica que o destino requisitado não foi encontrado.
3	Caminho não encontrado	Indica que o caminho requisitado não foi encontrado.
4	Outro	Indica qualquer outro tipo de erro.
5	Erro interno	Indica uma exceção que ocorreu no código do *gateway*. Esse erro existe exclusivamente por motivos de depuração.

Conforme discutido na Seção 4.2.3, a arquitetura também prevê o recurso de cache. Para auxiliar nos controles de cache, o modelo define os cabeçalhos a seguir.

Assinatura do cache (*opcional*): uma *string* de valor único que descreve o *payload* em cache. A sua implementação pode ser uma simples técnica de *hash* como MD5 (RIVEST, 1992). O seu uso foi baseado no cabeçalho *ETag* do HTTP: as mensagens de requisições enviam o valor armazenado e se a assinatura atual for diferente, uma nova resposta é enviada.

Expiração do cache (*opcional*): uma data e hora indicando quando o cache irá expirar. É um complemento à assinatura do cache. Após a expiração, uma nova requisição deverá ser feita.

Cache modificado (*opcional*): um valor lógico indicando que o conteúdo não foi modificado, isto é, a assinatura enviada é a mesma. É usado exclusivamente pelas respostas.

Por fim, as mensagens de resposta também possuem um cabeçalho especial para informar sobre erros na requisição, descrito a seguir.

Código de erro: quando esse cabeçalho está presente, a resposta é tratada como um erro que deverá ser tratado. Os códigos de erro definidos podem ser vistos na Tabela 4.2.

4.4 Subsistema de *Plugins*

O suporte a *plugins* é o elemento-chave para a extensibilidade do *gateway*. Esta seção é um guia sobre como esse suporte deve ser implementado.

Um *plugin* atua como cliente e servidor do protocolo implementado, para, respectivamente, fazer e atender requisições externas. No entanto, o servidor também é um cliente interno,

pois encaminha as requisições recebidas para o gerenciador de comunicação. Da mesma forma, o cliente é um servidor interno que atende aos pedidos do gerenciador.

As mensagens enviadas ou recebidas de fora do *gateway* são chamadas de **mensagens externas**. O formato dessas mensagens é o do protocolo lidado pelo *plugin*. No caso das enviadas ou recebidas do próprio *gateway*, essas são as **mensagens internas**. Elas utilizam o formato interno descrito na seção anterior.

Classificamos os *plugins* em dois tipos, cliente e servidor, na perspectiva da comunicação externa, onde:

***Plugin* Cliente** é responsável por atender às requisições internas e transformá-las em externas. As respostas externas, assim que chegam, são traduzidas para o formato interno e devolvidas ao *gateway*.

***Plugin* Servidor** conta com um servidor do protocolo implementado, onde, ao receber requisições, encaminha-as ao *gateway* como requisições internas. Quando as repostas internas chegam, devem ser devolvidas aos clientes externos, no formato específico do protocolo. O fluxo de dados sempre inicia em um *plugin* servidor quando um cliente externo o contata.

A relação de cada tipo de *plugin* com o *gateway* é ilustrada na Figura 4.4.

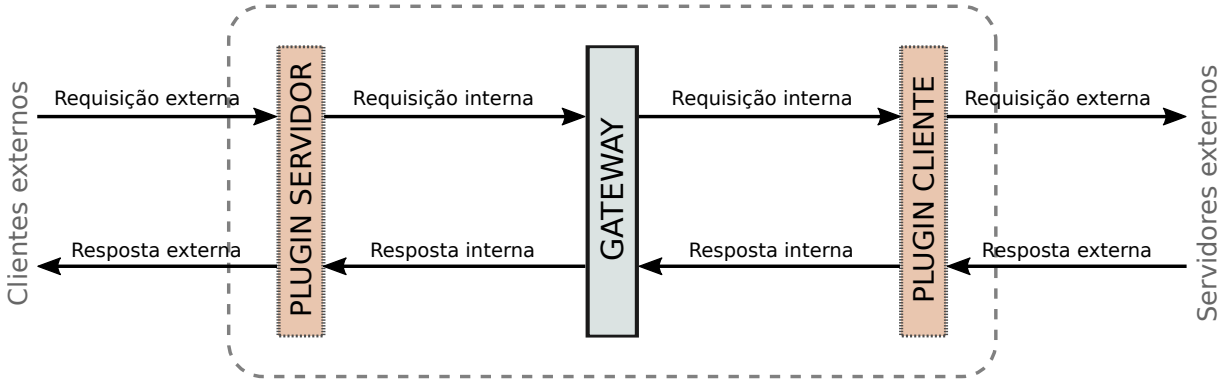


Figura 4.4 – Função de cada tipo de *plugin*.

4.4.1 Interface dos *Plugins*

Para implementar um *plugin*, de forma que o *gateway* possa interagir, a seguinte interface deve ser observada:

iniciar(): sem retorno – o *plugin* só deve começar a aceitar requisições após a chamada desse procedimento.

parar(): sem retorno – o *plugin* não deve mais aceitar requisições após a chamada desse procedimento.

obterProtocolo(): *string* – retorna um identificador do protocolo lidado pelo *plugin*. O nome não precisa ser de um protocolo existente, pode ser, por exemplo, "httpx", mas deve ser possível escrevê-lo em uma URI (*Uniform Resource Identifier*), conforme é explicado na Seção 4.5.

configurações(): tabela *hash* – o *plugin* deve oferecer entradas de configurações para definir itens como:

- a porta a ser ouvida pelo servidor;
- o tipo de soquete (TCP ou UDP);
- ativar ou desativar a parte cliente ou servidor.

tratarRequisição(Requisição): sem retorno – apenas o *plugin* cliente deve implementar esse procedimento para atender às requisições internas enviadas pelo *gateway*.

4.4.2 API do Gerenciador de Comunicação

Como único componente em que os *plugins* têm acesso, o gerenciador de comunicação deve oferecer, em contrapartida, uma API para que o *plugin* possa enviar ao *gateway* as requisições internas, enviar as respostas internas e também receber as respostas internas. A API que a implementação de *gateway* fornecer deverá ser, minimamente, uma variação da seguinte:

enviar(Requisição): inteiro – serve para o *plugin* servidor enviar as requisições internas ao *gateway*. É retornado o número de sequência da requisição que permite ao *plugin* obter a resposta.

receber(inteiro): Resposta – serve para o *plugin* servidor receber as respostas internas, passando o número de sequência recebido ao enviar.

enviar(Resposta): sem retorno – um procedimento para o *plugin* cliente enviar as respostas internas.

As assinaturas das funções e procedimentos tanto dessa API quanto das interfaces da seção anterior estão descritas em português, mas será comum as implementações utilizarem nomes em inglês. A Seção 4.7 aborda um protótipo experimental que foi implementado desse modo.

4.5 Identificação do Protocolo

É importante entender como uma requisição em um protocolo **A** é reconhecida pelo *gateway* como uma requisição a um protocolo **B**. Para isso, adotamos a URI (BERNERS-LEE et al., 1998) como o elemento que carrega a informação de roteamento entre os protocolos.

Usando como exemplo a mesma requisição HTTP-MQTT usada previamente e assumindo que o *gateway* está em `gw.me.com` e o servidor MQTT em `bk.you.org`, a seguinte URI acessada pelo cliente HTTP

```
http://gw.me.com/mqtt/bk.you.org/moisture
```

indica para o *gateway* que a requisição deverá ser encaminhada para o *plugin* MQTT (note `mqtt` no endereço) o qual é apto a requisitar `/moisture` do servidor em `bk.you.org`.

A Figura 4.5 ilustra a relação da URI com os elementos de interligação.

Figura 4.5 – Esquema de identificação do destino.

O processo de identificação dos elementos de destino da requisição é feito pelo Controlador de Interconexão (CIC), conforme mencionado na Seção 4.2.3.3. Mas veja que o CIC não recebe toda a URI acionada pelo cliente. No exemplo que acabamos de descrever, o *plugin* HTTP encaminha para o núcleo do *gateway* apenas `/mqtt/bk.you.org/moisture`, escrito no cabeçalho caminho. Quando o CIC receber essa requisição, escreverá o cabeçalho alvo com o valor `bk.you.org` e substituirá o valor do cabeçalho caminho por `/moisture`. Essas informações são o suficiente para o *plugin* de destino requisitar ao servidor informado.

É preciso ficar claro que cada *plugin* pode receber os parâmetros necessários à interconexão de formas diferentes, dependendo de como o protocolo implementado funciona ou mesmo como é feita a sua implementação. Como exemplo, se fôssemos usar o *gateway* para interligar clientes e servidores do mesmo protocolo[2]. Supondo uma comunicação HTTP-HTTP, para acessar o recurso `http://s1.com/luz` via *gateway* o cliente deverá requisitar de forma indireta pela URL `http://gw.me/http/s1.com/luz`. Essa indireção poderia ser resolvida se o *plugin* fosse implementado como um *proxy* HTTP[3].

Observe que nem todos os protocolos definem um esquema oficial de URI, no entanto, essa falta de padronização não impede de fazermos uma associação. O protocolo MQTT é um exemplo, que apesar de não definir um esquema, é perfeitamente possível considerar o tópico como o caminho, que é o único valor da URI que importa para o *gateway*.

[2] Do ponto de vista prático não faz muito sentido, mas é perfeitamente possível.

[3] Vale lembrar que uma implementação por *proxy* resolveria a indireção para o protocolo HTTP como destino, mas ainda ficaria o problema de qual seria a URL em destinos com protocolos diferentes.

Listagem 1: Exemplo de *listener* para alterar o modo de identificar o destino.

```
void converterEsquemaUri(Requisição req) {
    // Lê o valor do caminho dessa mensagem
    String caminho = req.cabeçalhos.get("caminho");

    // Processa o caminho
    String[] ss = caminho.split("/");
    String novoCaminho = String.format("/%s/%s/%s", ss[2], ss[1], ss[3]);

    // Escreve o novo caminho
    req.cabeçalhos.set("caminho", novoCaminho);
}
```

4.5.1 Alterando o Esquema de Identificação

O Controlador de Interconexão (CIC) é o componente responsável por ler o caminho da requisição interna e obter as informações do destino. Como a mensagem passa pelo Controlador de Entrada antes, serial possível adicionar um *listener* que permita modificar o esquema de identificação do *gateway*. A ideia básica é alterar o cabeçalho caminho para o padrão esperado pelo CIC.

Suponha que no novo padrão, o servidor de destino venha antes do protocolo. Dessa forma o cliente HTTP do exemplo anterior acessaria a seguinte URI:

`http://gw.me.com/bk.you.org/mqtt/moisture`

Um exemplo ilustrativo em Java de um *listener* que poderia fazer essa tarefa pode ser vista na Listagem 1.

Isso abre o leque para várias configurações diferentes. Poderíamos criar um índice permitindo diminuir o tamanho da URI, algo interessante para dispositivos restritos, permitindo, por exemplo, que ao acessar `/temperatura1` na verdade requisitar `/mqtt/bk.you.org/sensor1/temperatura`.

4.6 Fluxo da Comunicação Entre Protocolos

Esta seção descreve o fluxo da comunicação interprotocolo, desde a requisição externa, passando pelas mensagens internas, até a resposta externa, de forma a esclarecer melhor como os *plugins* utilizam as mensagens internas para interagir com o *gateway*.

Usamos como exemplo uma comunicação entre HTTP-MQTT. Supondo que um cliente HTTP quer contatar o servidor MQTT em `bk.you.org`, para alterar o conteúdo do tópico

`/s1/temperatura` e obter o valor de `/s1/bateria`. Como o cliente não pode se comunicar com MQTT, ele usa um *gateway* disponível em `gw.me.com`.

Na primeira ação, alterar o tópico, a comunicação HTTP-MQTT é ilustrada pelo diagrama de sequência da Figura 4.6. Descrevemos os passos da comunicação a seguir:

1. O cliente faz uma requisição HTTP usando o método `PUT` ao endereço:

   `http://gw.me.com/mqtt/bk.you.org/s1/temperatura`

2. O *plugin* servidor do HTTP recebe a requisição externa e transforma em uma mensagem de requisição interna sem o número de sequência e sem o cabeçalho obrigatório alvo, pois serão modificados depois pelo *gateway* ao passar pelo controlador de interconexão.

   O *payload* é copiado do próprio *payload* da mensagem HTTP e o cabeçalho caminho é preenchido com o valor `/mqtt/bk.you.org/s1/temperatura`. A operação será preenchida a depender da implementação do *plugin* HTTP, mas o método `PUT`, seguindo a especificação do HTTP, se encaixa melhor com a operação `ATUALIZAR`.

   A mensagem interna é enviada ao *gateway* que retorna o número de sequência da requisição que permitirá ao *plugin* receber a resposta.

3. O *gateway* lê a requisição interna e ao passar pelo controlador de interconexão:

   - o cabeçalho caminho é lido para identificar o protocolo de destino (MQTT) e o *host* destinatário da requisição;
   - em consequência, o cabeçalho alvo é modificado para `bk.you.org` e o caminho é reescrito com `/s1/temperatura`;

   Assim, a requisição interna pode ser encaminhada ao *plugin* MQTT.

4. O *plugin* cliente do MQTT lê a mensagem interna, contata o servidor em `bk.you.org` (obtido do alvo) e publica no tópico `/s1/temperatura` (obtido do caminho). Uma resposta interna de confirmação é devolvida ao *gateway*, com o número de sequência igual ao da requisição.

   A ação *publicar* é escolhida com base na operação solicitada na requisição interna. Nesse exemplo, o *plugin* MQTT mapeia `ATUALIZAR` para publicar.

5. O *gateway* aguarda pelo pedido da resposta que será feito pelo *plugin* HTTP, ao informar o número de sequência recebido no passo (2). Isso ocorre sem bloquear o *gateway*, que pode estar tratando outra requisição no mesmo momento.

6. Quando o *plugin* HTTP receber a resposta interna, ele a transforma em uma resposta no formato HTTP e devolve ao cliente.

Para a segunda ação, onde o cliente HTTP obtém o nível de energia de um nó MQTT, a sequência é praticamente a mesma, com poucas diferenças:

- No passo (1), o cliente HTTP usa o método `GET`.
- No passo (2), o método HTTP é traduzido para a operação `LER`.
- O passo (4) é o mais diferente. O *plugin* cliente MQTT mapeia a operação `LER` para assinar. Nesse caso, assim que a primeira resposta chegar, a assinatura é cancelada, e uma resposta interna é devolvida ao *gateway*, usando como *payload* o valor recebido do servidor externo.

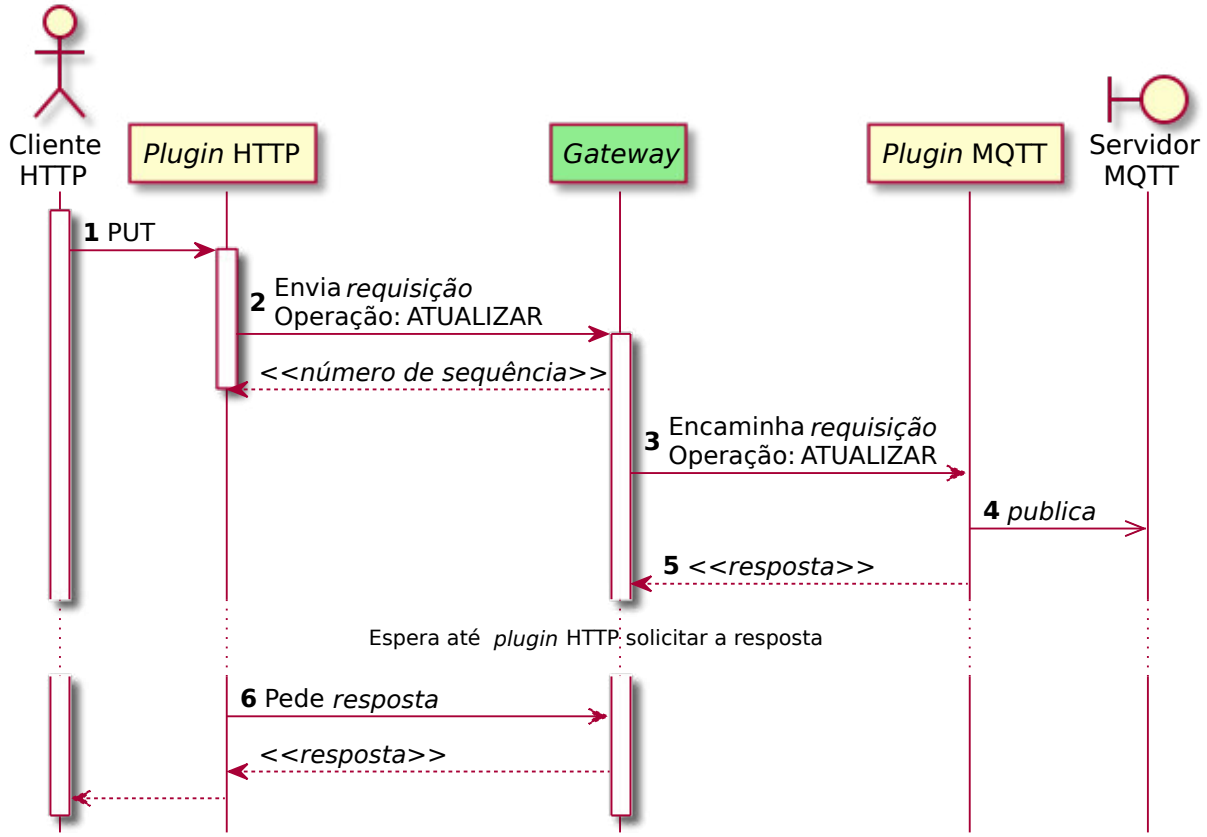


Figura 4.6 – Diagrama de sequência do fluxo interprotocolo.

Lembrando que todo esse processo vai depender inteiramente da implementação. Nesse exemplo, o *plugin* MQTT só mantém a assinatura se a operação fosse `OBSERVAR`, mas uma outra implementação poderia manter a assinatura por algum tempo para se outras requisições ao mesmo tópico aparecerem, não ser preciso uma nova conexão ao servidor externo.

Outro ponto importante a mencionar é que, como as requisições HTTP possuem um tempo máximo determinado pelo protocolo, enquanto outros protocolos como o MQTT não o possuem, então o *plugin* HTTP precisa ser implementado de forma a não ficar aguardando indefinidamente uma resposta interna. Isso vale para qualquer protocolo que sofram da mesma restrição.

4.7 Prova de Conceito

Foi desenvolvido um protótipo experimental para demonstrar a aplicabilidade da arquitetura proposta. O *gateway* foi implementado na linguagem Java 8 e, demonstrando a sua capacidade, interligou três protocolos: HTTP, CoAP e MQTT. A implementação ficou balanceada com os dois padrões de troca de mensagem, requisição-resposta (HTTP e CoAP) e publicar-assinar (MQTT).

4.7.1 Mapeamentos Utilizados

As tabelas 4.3 e 4.4 apresentam os mapeamentos que utilizamos nos *plugins*. A primeira tabela indica qual o tipo de requisição cada *plugin* cliente fará a um servidor externo quando receber uma requisição interna com as dadas operações. A segunda mostra o que ocorre no sentido contrário, qual a operação interna será escolhida para cada requisição recebida de um cliente interno. Note que nem todos os mapeamentos da Tabela 4.4 estão preenchidos. Isso não caracteriza um problema, já que os *plugins* servidor lidam com as requisições externas e pode escolher a melhor forma de atendê-las. Já não se pode dizer o mesmo dos *plugins* cliente, já que lidam com requisições internas, que pode ser de qualquer operação.

Os mapeamentos das operações básicas para os protocolos HTTP e CoAP foram feitos seguindo as diretrizes das suas especificações (FIELDING; RESCHKE, 2014; SHELBY; HARTKE; BORMANN, 2014), respeitando o quanto possível as semânticas originais.

Nas operações `OBSERVAR` e `DESOBSERVAR`, a implementação de HTTP é um simples *polling*, i.e. fazendo requisições contínuas ao servidor solicitado num dado intervalo. Não é uma boa estratégia, principalmente falando de IoT, mas o protocolo HTTP não especifica respostas assíncronas. Na verdade, existem algumas técnicas muito utilizadas (LORETO et al., 2011) que consideramos implementar, mas não há como detectar se o servidor HTTP solicitado implementa alguma delas.

O CoAP se diferencia nisso com relação ao HTTP, já que ele estende a requisição `GET` com a opção *Observe* (HARTKE, 2015). Quando um servidor CoAP recebe esse tipo de pedido, o cliente é adicionado à lista de observadores e o servidor enviará uma resposta ou tantas outras até que um dos lados cancele a observação. Para que funcione, o servidor deverá implementar essa extensão e aceitar pedidos com *Observe* no dado recurso. Desse modo, a nossa implementação da operação `OBSERVAR` no *plugin* cliente do CoAP funciona assim: (1) tenta requisitar com *Observe*, (2) se não é possível, recua para o simples *polling*.

O protocolo MQTT (BANKS; GUPTA, 2014) é menos rico, de maneira que só existem três ações: publicar, assinar e cancelar a assinatura. As operações `OBSERVAR` e `DESOBSERVAR` são facilmente mapeadas para assinar e cancelar a assinatura. Já o mapeamento das outras são mais difíceis, uma vez que a requisição interna possui seis operações.

Tabela 4.3 – Mapeamentos das operações internas para os *plugins* do tipo cliente.

Operação interna	Método HTTP	Método CoAP	Ação MQTT
`CRIAR`	`POST`	`POST`	Publicar (retida)
`ATUALIZAR`	`PUT`	`PUT`	Publicar (não retida)
`REMOVER`	`DELETE`	`DELETE`	Publicar (retida e sem *payload*)
`LER`	`GET`	`GET`	Assinar (uma resposta)
`OBSERVAR`	`GET` (em *polling*)	`GET` (*Observe* ou em *polling*)	Assinar
`DESOBSERVAR`	Cancelar *polling*	Cancelar *Observe* ou o *polling*	Cancelar assinatura

Tabela 4.4 – Mapeamentos dos *plugins* do tipo servidor para as operações internas.

Operação interna	Método HTTP	Método CoAP	Ação MQTT
`CRIAR`	`POST`	`POST`	Publicar (retida)
`ATUALIZAR`	`PUT`	`PUT`	Publicar (não retida)
`REMOVER`	`DELETE`	`DELETE`	Publicar (retida e sem *payload*)
`LER`	`GET`	`GET`	–
`OBSERVAR`	–	`GET` (*Observe*)	Assinar
`DESOBSERVAR`	–	Cancelar *Observe*	Cancelar assinatura

Assim, para garantir que o *plugin* cliente aceite todos os tipos de operações internas, nos aproveitamos do recurso do protocolo MQTT chamado publicação retida. Por padrão, após a publicação ser repassada aos assinantes do tópico, o *broker* remove o conteúdo, de forma que futuros assinantes não o receberão. Publicar de forma retida muda isso, pois o conteúdo é mantido até que uma nova publicação retida seja feita.

É especificado que uma publicação retida com o *payload* vazio remove o conteúdo armazenado (BANKS; GUPTA, 2014), assim a operação `REMOVER` foi mapeada para essa ação. As operações `CRIAR` e `ATUALIZAR` foram mapeadas respectivamente para a publicação retida e a publicação não retida, para que todas as operações fossem utilizadas, mas não há realmente esses significados para o protocolo MQTT.

No caso da operação `LER`, a ação do *plugin* cliente é (1) assinar o tópico no destino especificado, (2) esperar uma resposta e (3) assim que a primeira resposta chegar, cancelar a assinatura para não receber mais um dado que é inútil.

Tabela 4.5 – Mapeamentos de alguns cabeçalhos opcionais.

Interno	HTTP	CoAP	MQTT
Tipo de conteúdo	*Content-Type*	*Content-Type*	–
Tipos esperados	*Accept*	*Accept*	–
QoS	–	Tipo da mensagem	QoS

4.7.1.1 Mapeamentos Específicos

Os mapeamentos das operações internas são os mais importantes. Porém também fizemos os mapeamentos de alguns dos cabeçalhos opcionais. A Tabela 4.5 exibe como os cabeçalhos foram mapeados de acordo com os protocolos.

Não é surpresa o mapeamento utilizado de tipo de conteúdo e tipos esperados nos protocolos HTTP e CoAP, visto que esses cabeçalhos internos foram inspirados neles. Já no protocolo MQTT, não há nenhuma opção que indique o tipo do *payload* enviado ou recebido, portanto não foi mapeado.

No caso do cabeçalho QoS, que deve indicar o quão confiável será a entrega da requisição, o HTTP não possui esse tipo de recurso, mas é nativo do MQTT, com três níveis de QoS, em ordem crescente de confiabilidade. Já CoAP, embora não nomeie assim, o tipo da mensagem de requisição CoAP faz o mesmo efeito de QoS, onde cada mensagem enviada pode ter dois tipos: confirmável (mapeado para 1) e não confirmável (mapeado para 0).

4.7.2 *Framework* de Desenvolvimento de *Plugins*

Um pequeno *framework* foi desenvolvido para facilitar a criação e integração dos *plugins* baseada na API vista na Seção 4.4. Os elementos do *framework* podem ser vistos no diagrama de classes da Figura 4.7.

Para criar um novo *plugin* o desenvolvedor apenas precisa implementar a interface `PluginClient`, para o tipo cliente, ou a interface `PluginServer`, para o tipo servidor, ou mesmo ambas, para um *plugin* com cliente e servidor. O gerenciador de comunicação, representado por `CommunicationManager` no diagrama, é o responsável por registrar e iniciar cada *plugin*.

Note os métodos `setUp(ReplyLink)` e `setUp(RequestLink)`. De forma a desacoplar completamente os *plugins* da implementação do gerenciador de comunicação, cada cliente recebe uma instância de `ReplyLink` para enviar respostas internas, enquanto cada servidor recebe uma instância de `RequestLink` para enviar requisições internas.

Quando um *plugin* servidor envia uma requisição com `RequestLink::send`, uma instância de `FutureReply` é retornada. Essa é uma especialização da interface `Future`, do

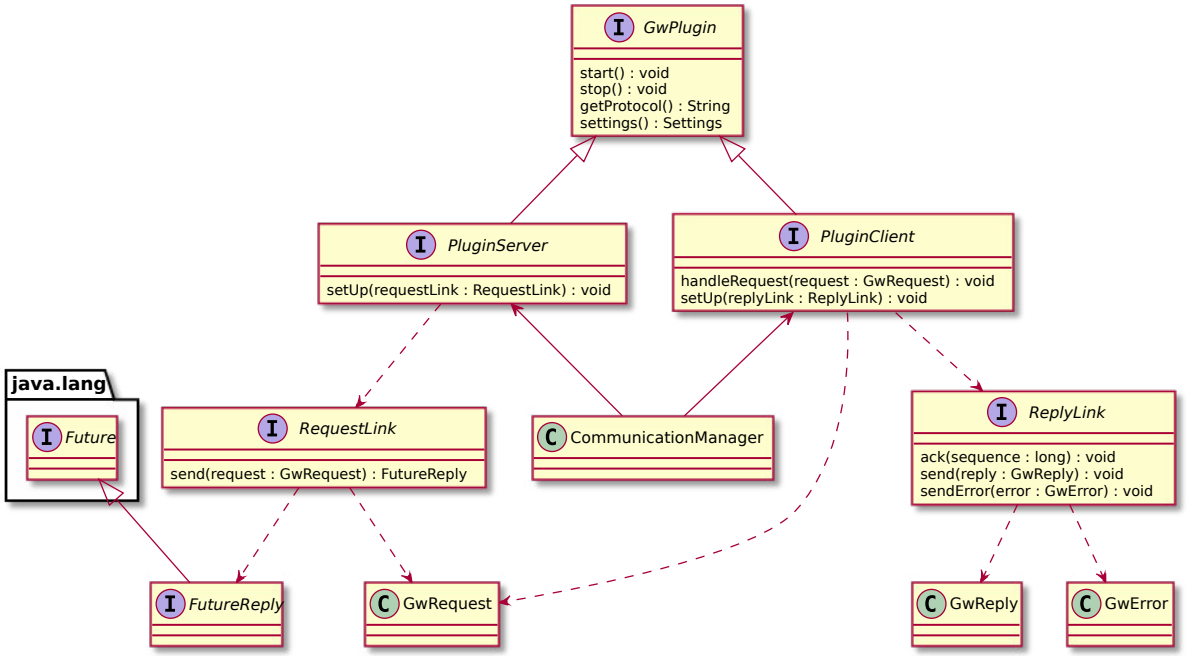


Figura 4.7 – Diagrama de classes da implementação do subsistema de *plugins*.

pacote `java.util.concurrent`, que permite ao *plugin* obter a resposta de forma síncrona ou assíncrona. Veja que na descrição da API do gerenciador de comunicação, na Seção 4.4.2, é sugerido que o retorno imediato de uma requisição seja o número de sequência da mensagem, para obter a resposta em outro momento. No entanto, aproveitando melhor os recursos da linguagem, adotamos esse *design*, facilitando mais para o desenvolvedor.

O *plugin* cliente pode usar três métodos em `ReplyLink`, onde:

- `ack(long)` permite enviar uma resposta interna vazia, apenas indicando o sucesso da requisição interna recebida. Esse método é um facilitador para enviar respostas quando estas não exigem um *payload*.
- `send(GwReply)` permite enviar uma resposta interna completa, com sequência, cabeçalhos e *payload*.
- `sendError(GwError)` permite enviar respostas internas que representam erros. O tipo `GwError` é igual a `GwReply`, mas adiciona um campo para o código do erro.

A cada *plugin* é garantida uma *thread* para se comunicar com o *gateway*. Dessa forma, como cada um tem a sua própria fila de requisição, um *plugin* mal implementado não bloqueia os outros.

Adicionalmente, é importante saber que o gerenciador de comunicação gera uma implementação dedicada de `RequestLink` e `ReplyLink` para cada *plugin* registrado numa adaptação do padrão de projeto *Reactor* (BUSCHMANN; HENNEY; SCHMIDT, 2007). Isto quer dizer

que a comunicação entre o *gateway* e os *plugins* não usam sincronização de *threads* para a troca de dados, reduzindo os problemas decorrentes do seu uso.

4.7.3 Comentários

4.7.3.1 Sobre os Softwares Utilizados

Os *plugins* foram implementados com o auxílio dos seguintes *softwares* livres:

- *Apache HTTP Components* (APACHE..., 2015): para ambos os tipos de *plugin* do protocolo HTTP.
- *Californium CoAP Framework* (CALIFORNIUM..., 2016): para ambos os tipos de *plugin* do protocolo CoAP.
- *Paho Client for Java* (PAHO..., 2015): para o *plugin* cliente do MQTT.
- *Moquette Broker* (MOQUETTE..., 2015): para o *plugin* servidor do MQTT.

4.7.3.2 Sobre os *plugins* HTTP e CoAP

No total desenvolvemos três *plugins* HTTP. O primeiro foi baseado no *framework Netty* (NETTY, 2015), mas logo foi descartado devido à quantidade de *bugs*. O segundo, usando a biblioteca *NanoHTTPD* (NANOHTTPD, 2015), funcionou bem o suficiente para demonstrar o funcionamento do *gateway*, mas tinha problemas graves de desempenho, sendo substituído por último pelo já mencionado *Apache HTTP Components*.

A experiência nas trocas da implementação desse *plugin* demonstrou a eficiência do sistema no quesito extensibilidade, já que é possível substituir a implementação sem algum ajuste necessário ao núcleo do *gateway*.

Não houve grandes dificuldades na implementação do protocolo CoAP. Já que a sua semântica é tão parecida com HTTP, o *plugin* foi quase uma cópia desse, inclusive os mapeamentos foram os mesmos. Apenas alguns ajustes foram necessários para se adaptar à biblioteca *Californium* e implementar a opção *Observe*.

Uma nota importante para HTTP e CoAP é que o *gateway* não permite uma experiência completa do padrão REST (FIELDING; TAYLOR, 2002). A principal razão para isso é porque nem todas as regras necessárias às interações REST, como os códigos de respostas das mensagens, podem ser mapeados, para a mensagem interna, devido a limitações que impomos ao modelo de mensagem. Deixamos a compatibilidade com REST como um trabalho futuro.

4.7.3.3 O Desafio do Servidor MQTT Interno

Enquanto as bibliotecas HTTP e CoAP são, em geral, feitas para criar servidores altamente customizados, as bibliotecas MQTT vêm como servidores prontos que impedem personalizar o seu comportamento. O desafio aqui está em que, para habilitar a interconexão desse

protocolo, é preciso que o servidor MQTT interno, isto é, o usado na implementação do *plugin*, desvie da especificação do protocolo, nos seguintes pontos:

1. O servidor não deverá encaminhar diretamente as publicações recebidas aos assinantes. Antes disso, as mensagens de publicação deverão ser encaminhadas ao *gateway*, para então, quando uma nova resposta for recebida do *gateway*, encaminhá-la aos assinantes.
2. O valor recebido pelos assinantes não será necessariamente o mesmo valor da última publicação.
3. Uma publicação, mesmo que confirmada ao publicador, pode nunca ser enviada aos assinantes.

A Figura 4.8 ilustra o desafio. Nela, um cliente MQTT **A** assina um tópico que representa um recurso HTTP, e outro cliente **B** publica nesse tópico. Se o servidor MQTT funcionasse como manda a especificação do protocolo, o valor publicado por **B** seria repassado imediatamente para **A** (fluxo 1).

Agora observe o fluxo 2, que representa um servidor MQTT modificado. Nele, o valor repassado aos assinantes é na verdade o retornado pelo servidor HTTP, porém esse valor pode ser diferente do valor publicado por **A** devido a algum processamento no servidor HTTP. Isso não ocorre no fluxo 1, que sempre repassa aos assinantes exatamente o valor que recebe dos publicadores.

Utilizamos para implementar o *plugin* servidor o *Moquette Broker*. Para fazer ele funcionar de acordo com os três itens listados anteriormente, precisamos estudar seu código para saber onde incluir um interceptador que permitiria alterar o seu fluxo normal. Uma parte das nossas alterações foram incorporadas ao projeto original e algumas específicas estão sendo mantidas separadamente em https://github.com/gothings/moquette/tree/interceptor.

4.7.3.4 Limitações do Protótipo

Em geral, o funcionamento da implementação foi satisfatória. Mas, no momento de escrita dessa dissertação, alguns problemas não foram resolvidos. Destacamos os seguintes:

- A operação `OBSERVAR` não faz exatamente o que deveria. O *gateway* deveria apenas repassar uma nova resposta quando for diferente da anterior, porém nessa implementação qualquer resposta que chega é encaminhada.
- Cache não implementado. Embora, ao receber uma requisição o controlador de interconexão verifique a presença de cache, o método responsável está vazio, isto é, sempre retorna nada, indicando que não há cache.
- O *gateway* não se recupera bem de conexões interrompidas com os clientes externos, deixando requisições abertas com um servidor desnecessariamente, visto não haver alguém para encaminhar a resposta.

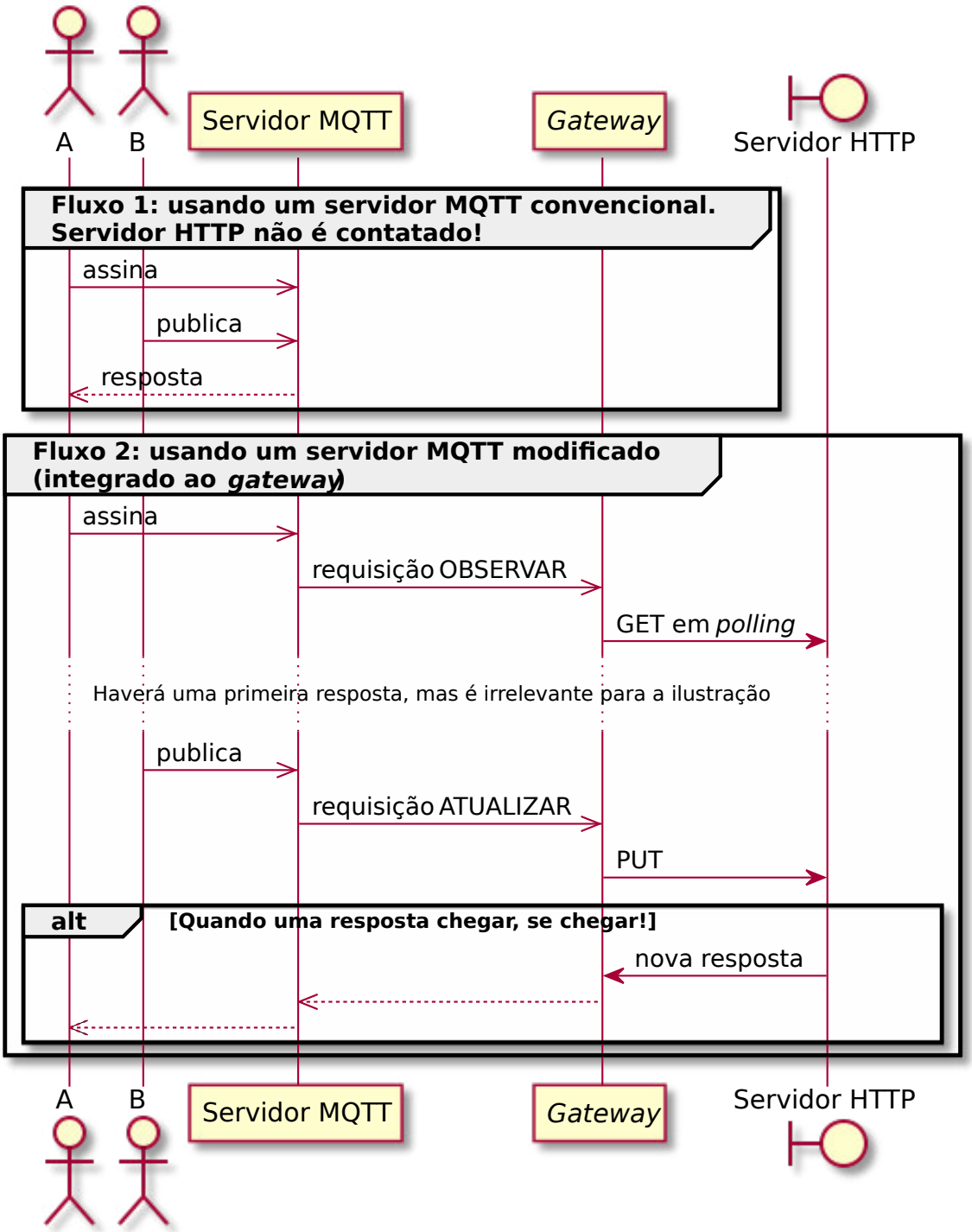


Figura 4.8 – Servidor MQTT convencional e modificado.

4.7.4 Disponibilidade

Todo o código-fonte está disponível no site https://github.com/gothings/gothings-gateway sob uma licença livre, permitindo seu uso, modificação e redistribuição.

4.8 Discussão

Um dos pontos-chave da arquitetura proposta é permitir a criação de *plugins* de uma forma amigável ao desenvolvedor. Ao capacitar o *gateway* com a possibilidade de utilizar bibliotecas que já implementam os protocolos, o desenvolvedor não precisa se envolver em quase nenhum momento com a lenta programação de soquetes. Pois, se já existe uma boa implementação de um dado protocolo, não há razão para implementá-lo do zero.

A abordagem de interoperabilidade do *gateway* é a de ponte indireta. Um problema comum nesse tipo de abordagem é a perda da semântica de um ou de ambos protocolos. Não achamos que a arquitetura GoThings sofra com esse problema porque o foco da solução são nos padrões de interação da IoT que são usualmente muito simples e específicos, conforme discutido na Seção 2.1.

Outra questão importante é o lugar onde o *gateway* é usado. Acreditamos que a melhor posição seja nas bordas da rede, pois ficará perto dos clientes locais, habilitando-os à comunicação interprotocolo localmente, mas também, sendo uma porta de saída para essa comunicação com as redes externas. Além disso, o *gateway* poderá servir de porta de entrada para as redes externas acessarem os servidores locais.

5 Avaliação e Resultados

Este capítulo apresenta a nossa avaliação quanto aos requisitos da arquitetura expostos no capítulo anterior e uma avaliação de desempenho do *gateway* implementado.

5.1 Avaliação dos Requisitos

Os requisitos definidos para a arquitetura foram: (1) interoperabilidade, (2) extensibilidade, (3) generalidade, (4) configurabilidade e (5) adequação a dispositivos restritos. A avaliação de cada um dos requisitos é discutida a seguir.

5.1.1 Avaliação da Interoperabilidade

Para a avaliação da interoperabilidade, verificamos se os três protocolos com *plugins* implementados conseguiram trocar dados corretamente e verificamos o que não pôde ser mapeado e o possível impacto disso.

Na primeira verificação, fizemos três testes onde em cada um definimos um servidor de um dos protocolos, com clientes dos outros dois protocolos fazendo requisições ao *gateway*.

No primeiro teste, um servidor HTTP foi preparado enquanto requisições com clientes CoAP e MQTT foram feitas ao *gateway*, que repassou a esse servidor. Nos outros dois testes, alternamos o servidor alvo e os clientes usados. Verificamos que todas as requisições e respostas chegaram, traduzidas da forma esperada, conforme os mapeamentos definidos no capítulo anterior. Também testamos de clientes para servidores do mesmo protocolo, com as requisições e respostas passando pelo *gateway*, verificando o mesmo resultado. Os cenários de testes são ilustrados pela Figura 5.1.

Os testes entre o mesmo protocolo também foram utilizados como insumo para a segunda verificação. Apuramos o que não foi possível mapear, seja por defeito da implementação ou limitação do modelo de mensagem definido pela arquitetura. Essa verificação consistiu em fazer requisições passando pelo *gateway*, e verificar se todas as semânticas do protocolo foi mantida, tanto na própria requisição recebida pelo servidor, quanto na resposta devolvida ao cliente.

Os resultados para o MQTT foram muito bons. Sendo um protocolo muito simples, com poucas opções, o modelo de mensagem permitiu manter toda a sua expressividade, comprovado pelos resultados obtidos nos testes feitos com o protótipo. As requisições MQTT testadas foram:

1. publicar com a opção retida desativada;
2. publicar com a opção retida ativada;
3. assinar;

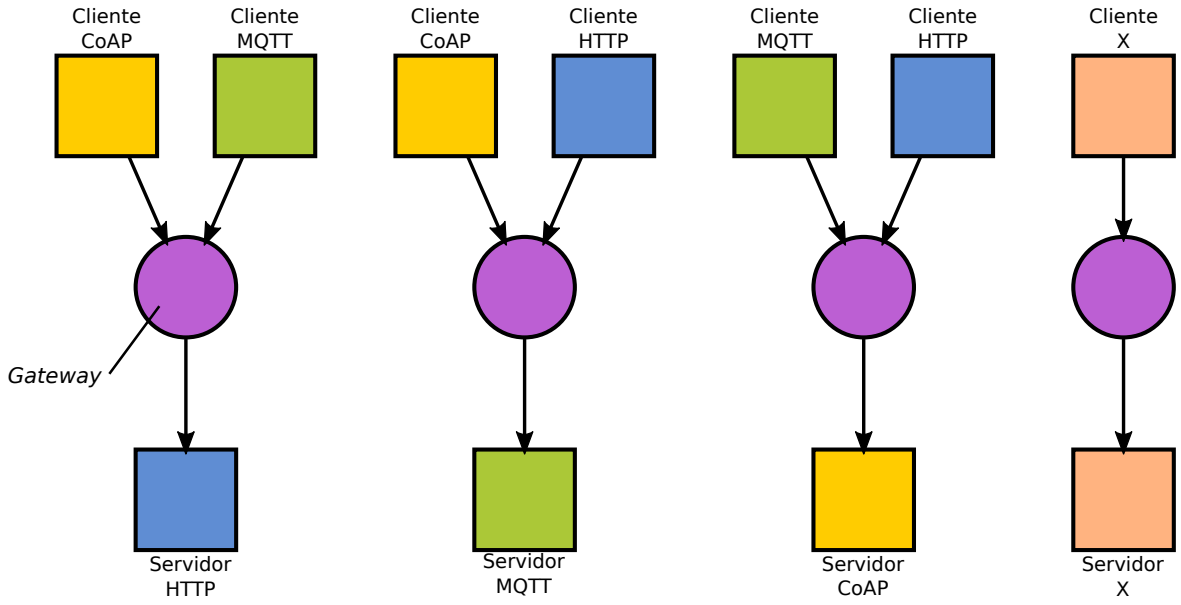


Figura 5.1 – Cenários usados na avaliação da interoperabilidade.

4. cancelar assinatura.

Conforme mencionado na Seção 2.2.2, o protocolo CoAP foi projetado para ser mapeável para HTTP. Mas, sendo o CoAP feito para ser usado por dispositivos restritos, ideal para a IoT, as suas opções são bem pequenas comparando com HTTP, que é bastante rico, resultado de décadas de refinamento. Em vista disso, restringimos a nossa avaliação do protocolo HTTP aos recursos disponíveis no CoAP.

Dito isso, os mapeamentos tanto entre HTTP-HTTP e CoAP-CoAP foram satisfatórios dentro do que esperávamos para esse trabalho. No entanto, não pode-se deixar de mencionar os seguintes pontos:

1. Todas as respostas HTTP/CoAP seguem um código de estado, que indica sucesso ou erro na requisição. Já o modelo de mensagem da arquitetura foi projetado com as respostas enviando um código apenas em erros, de forma que a falta de um código indica o sucesso. Adicionalmente, a quantidade de códigos de erro desses protocolos é bem maior.
2. O modelo de mensagem define que apenas as requisições com as operações `LER` e `OBSERVAR` retornam um *payload*, ao passo que HTTP/CoAP também prevê *payload* para a requisição mapeada para a operação `CRIAR`.

O item 2 é bastante relevante, pois, como brevemente mencionado na Seção 4.7.3.2, não foi possível implementar uma interação seguindo o padrão de interação REST, e essa limitação do modelo de mensagem é um dos motivos. Porém não pode-se afirmar que apenas solucionando esse ponto habilitaria interações com esse padrão.

Mesmo com esses pequenos problemas, consideramos a capacidade de interoperação conseguida pelo *gateway* um sucesso.

5.1.2 Avaliação da Extensibilidade

Conforme afirmado antes, GoThings faz uso da abordagem de ponte indireta porque focamos na facilidade de plugar qualquer número de protocolos previamente desconhecidos.

O subsistema de *plugins* é o principal responsável por essa capacidade. De fato, habilitar um novo protocolo em um *gateway* é só uma questão de desenvolver e instalar um novo *plugin*.

Na avaliação desse requisito, provamos a capacidade de estender o *gateway* através dos *plugins* desenvolvidos, descritos no capítulo anterior. Todos eles puderam ser desenvolvidos de forma independente, sem a necessidade de alterar o código do núcleo do *gateway*.

5.1.3 Avaliação da Configurabilidade e Generalidade

Os *listeners* dentro dos controladores de entrada e saída são a chave para a configurabilidade. Eles permitem uma configuração mais granular de um *gateway*. Com eles, como exemplo, poderíamos instruir o *gateway* para assegurar que a resposta seja no formato JSON (*JavaScript Object Notation*) se a requisição veio de um dispositivo restrito. Ou poderíamos programar um ou mais *listeners* para adicionar a um *gateway* recursos de interoperabilidade semântica.

Ao prover tais mecanismos que podem alterar o comportamento de um *gateway*, a configurabilidade dá suporte para a generalidade. O modelo de mensagem também se provou genérico o suficiente, conforme confirmado pela avaliação de interoperabilidade. Neste sentido, é importante lembrar que o conceito de generalidade deste artigo é no contexto dos problemas da IoT.

Embora esses requisitos não tenham sido provados na prática, os argumentos fazem sentido.

5.1.4 Avaliação da Adequação a Dispositivos Restritos

O cache no controlador de interconexão é o elemento-chave que faz a arquitetura GoThings apta aos dispositivos restritos. Um cache permite que um *gateway* reduza o encaminhamento de requisições e garanta que um valor seja sempre retornado, mesmo quando os dispositivos requisitados estejam em suspensão.

No capítulo anterior, pudemos ver como o cache pode reduzir o *overhead* de comunicação. Em geral, ao evitar requisições a dispositivos restritos, o *gateway* torna-se adequado a eles.

Esse requisito também não foi provado na prática, pois não chegamos a implementá-lo.

5.2 Avaliação de Desempenho

Embora o foco desse trabalho não tenha sido o desempenho da solução, fizemos uma avaliação da implementação de prova de conceito para verificar o impacto da comunicação via *gateway* no desempenho.

5.2.1 Cenário

O cenário avaliado são de clientes de cada protocolo, fazendo requisições a um servidor do seu próprio protocolo, mas através do *gateway*. Para poder avaliar o impacto da presença do *gateway* fazendo o intermédio da comunicação, também foram verificadas as requisições de forma direta, isto é, dos clientes direto para o servidor.

A medida usada foi do tempo de execução em que cada programa cliente levou para iniciar, fazer a requisição, receber a resposta e encerrar.

A execução dos testes foram realizados em um *laptop* Dell Inspiron 1545, equipado com o processador Intel Core 2 Duo T660 (2.20GHz, 2MB cache), 4GB RAM DDR2, utilizando o sistema Ubuntu Linux 15.10 na versão 64 *bits* com o *kernel* Linux 4.2.0.

5.2.2 Ferramentas

O comando `time` (TIME(1), 2015) do *shell* do Linux: usado para medir o tempo de execução que cada cliente utilizou para contatar o servidor e receber uma resposta. Ele foi usado nas medições das requisições cliente-servidor e cliente-*gateway*-servidor.

Para os clientes e servidores usados nos testes, foram usados os seguintes:

- HTTP: foi utilizado o cliente de linha de comando *curl* (CURL. . . , 2016) e para o servidor, foi usado o *SimpleHTTPServer* da biblioteca padrão do Python 2.7.
- CoAP: foi utilizado o cliente provido pela *libcoap* (BERGMANN, 2016), enquanto o servidor foi desenvolvido em Java usando a biblioteca *Californium* (CALIFORNIUM. . . , 2016).
- MQTT: o cliente foi feito em Python com a biblioteca *Paho Client for Python* (PAHO. . . , 2015) e como servidor foi usado o *Moquette Broker* (MOQUETTE. . . , 2015).

5.2.3 Experimento

Como o objetivo foi de verificar o impacto da comunicação através do *gateway*, todos os servidores foram configurados para entregar apenas dados estáticos. Conforme Elmangoush et al. (2015), a maioria das mensagens trafegadas na IoT possuem de poucos *bytes* a alguns *kilobytes*, de modo que o tamanho dos *payloads* usados nos testes variaram entre 1 B até

64 KB. Os conteúdos usados foram de valores aleatórios gerados pelo arquivo especial do Linux `/dev/urandom`.

Assim, os servidores HTTP e CoAP foram configurados para entregar esses arquivos dispostos em um diretório. Já no servidor MQTT, foram feitas inicialmente publicações retidas para cada um dos arquivos gerados, necessário para não ser preciso fazer a sincronização entre publicador e assinante no momento dos testes. Essas definições foram o suficiente para verificar-se exatamente quanto tempo todo o processo levou.

Os clientes foram configurados para fazer uma requisição que mapeasse para a operação interna `LER` e fechasse a conexão logo após receber a resposta. Assim, os clientes HTTP e CoAP usaram o método `GET` e o cliente MQTT usou a ação de assinar.

Inicialmente o programa cliente de cada protocolo foi acionado para requisitar diretamente aos servidores. Em seguida, os clientes requisitaram a mesma informação, mas através do *gateway*. Para cada tamanho de *payload* foram feitas 100 requisições e calculada a média. Mas antes de medir os tempos, executamos algumas vezes cada cliente para poder aquecer o cache do processador, de forma que os resultados tivessem menos interferência. É importante deixar claro que todo o experimento foi realizado em *localhost*.

5.2.4 Resultados

Os resultados do tempo de execução nas requisições para todos os tamanhos de *payload* podem ser vistos nos gráficos das figuras 5.2, 5.3 e 5.4, para, respectivamente, as requisições HTTP-HTTP, CoAP-CoAP e MQTT-MQTT. Os valores exibidos são da média aritmética com intervalo de confiança de 95%.

As tabelas 5.1, 5.2 e 5.3 mostram as análises do impacto no desempenho pelo fato das requisições passarem pelo *gateway* antes de chegar ao destino. Todas as três tabelas possuem a mesma estrutura explicada a seguir. A segunda coluna indica a média do tempo de execução medido para as requisições feitas pelo cliente diretamente ao servidor. Na terceira coluna é mostrado o mesmo, mas para as requisições feitas através do *gateway*. A quarta e a quinta coluna mostram, respectivamente, o tempo a mais e o impacto pelas requisições terem sido feitas via *gateway*.

É importante elucidar que os valores coletados não devem ser usados para verificar qual o melhor protocolo. As medidas refletem apenas os tempos de execução levados com o uso das implementações utilizadas nos clientes e servidores de testes, bem como das implementações dos *plugins* do *gateway*.

5.2.5 Discussão dos Resultados

O aumento no tempo de execução pode ser prejudicial para aplicações críticas, mas é quase imperceptível para grande parte das aplicações para a IoT. Conforme indica Elmangoush

et al. (2015), as aplicações podem tolerar até horas de latência dependendo do serviço oferecido, com exceção das aplicações *eHealth* (utilizadas em aparelhos hospitalares), cuja ineficiência na comunicação pode causar perdas de vidas.

Dito isso, a seguir, discutimos sobre os resultados de cada protocolo.

5.2.5.1 De HTTP para HTTP

O impacto no desempenho em requisições HTTP-HTTP variou entre 51% a 63% mais lento, sendo que o tempo de execução a mais ficou em torno de apenas 7 milissegundos, com um aumento perceptível a partir do *payload* de 32 KB. O tempo extra chegou a, em média, 14,64 milissegundos na requisição com *payload* de 64 KB.

Consideramos esses valores bem aceitáveis, mas cabe comentar que um aumento muito grande no tempo da requisição pode quebrar a comunicação HTTP, pois seus clientes definem um tempo máximo de espera. Porém, geralmente, os clientes HTTP possuem uma tolerância bem grande, esperando vários segundos pela resposta, de forma que o tempo a mais causado pelo *gateway* não deve causar problemas considerando *payloads* até 64 KB.

Os tempos extras experimentados na comunicação HTTP-HTTP foram maiores que dos outros protocolos. O motivo é que, diferente de CoAP e MQTT, o protocolo HTTP não é binário, de forma que a requisição extra feita pelo *gateway* ao servidor de destino causa esse aumento. Outro fator impactante é que a implementação do *plugin* de cliente HTTP não mantém uma conexão aberta com o servidor de destino, sendo necessário abrir uma nova conexão a cada requisição. No caso dos experimentos, como o servidor é sempre o mesmo, uma conexão permanente reduziria o tempo total da requisição.

5.2.5.2 De CoAP para CoAP

Apesar do impacto no desempenho causado pelo *gateway* como intermediário da comunicação ter sido o maior entre todos, chegando a 113% mais lento (mais que o dobro do tempo), o protocolo CoAP obteve bons resultados, com o tempo extra, na maioria dos tamanhos de *payload* em torno de 1,5 milissegundos. Lembrando que, sendo implementado por UDP, o cliente CoAP não precisou realizar os processos de conexão e desconexão.

Pela sua especificação (SHELBY; HARTKE; BORMANN, 2014), o tamanho recomendado da mensagem no CoAP é de até 1024 *bytes*, para caber em um datagrama UDP. Após esse tamanho a mensagem gera muita fragmentação, afetando o desempenho. Os resultados obtidos refletem isso, o tempo extra experimentado em *payloads* de até 1 KB chega a no máximo 2,18 milissegundos, aumentando consideravelmente em tamanhos maiores que isso.

5.2.5.3 De MQTT para MQTT

Os melhores resultados foram obtidos nas requisições MQTT-MQTT, tanto em questão do impacto na comunicação, 17% mais lenta, quanto no tempo extra necessário para o *gateway* fazer o intermédio, em torno de 5 milissegundos para todos os *payloads*. Os resultados só foram um pouco ruins em *payloads* muito pequenos, de até 2 *bytes*, chegando a 10,57 milissegundos em 1 *byte*.

É provável que os bons resultados do MQTT sejam devidos à implementação do *plugin* de cliente, que apesar de assinar um tópico no servidor de destino a cada requisição recebida, mantém uma conexão aberta com este, eliminado o processo de *handshake* do TCP.

5.2.5.4 Comentário

O aumento no tempo de execução era inevitável, mas o impacto no desempenho causado pelo intermédio do *gateway* é aceitável, considerando os tamanhos de *payload* usados no experimento, conforme discutido nas seções anteriores. De modo que, se o *gateway* estiver próximo ao cliente ou servidor interligado, o aumento no tempo em que a resposta é devolvida não será tão impactante.

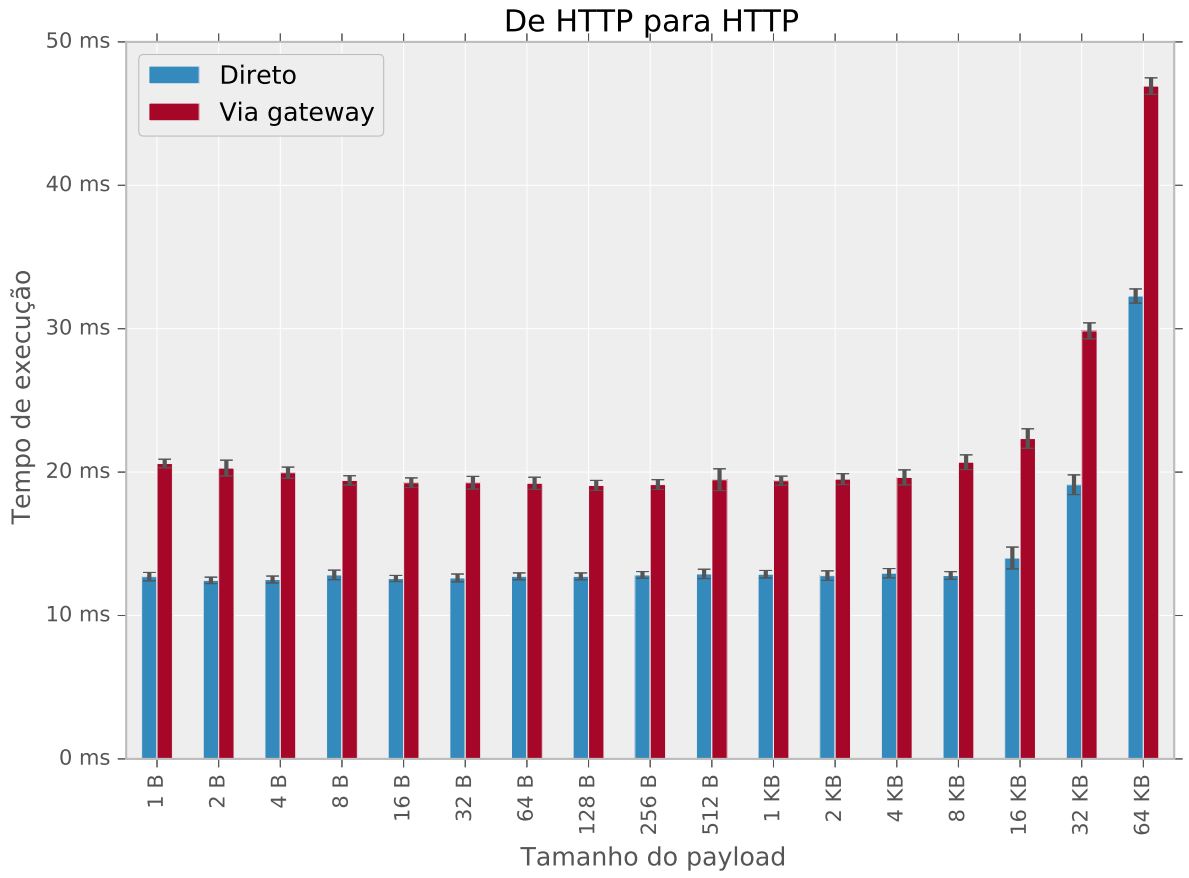


Figura 5.2 – Resultados do tempo de execução para requisições HTTP-HTTP.

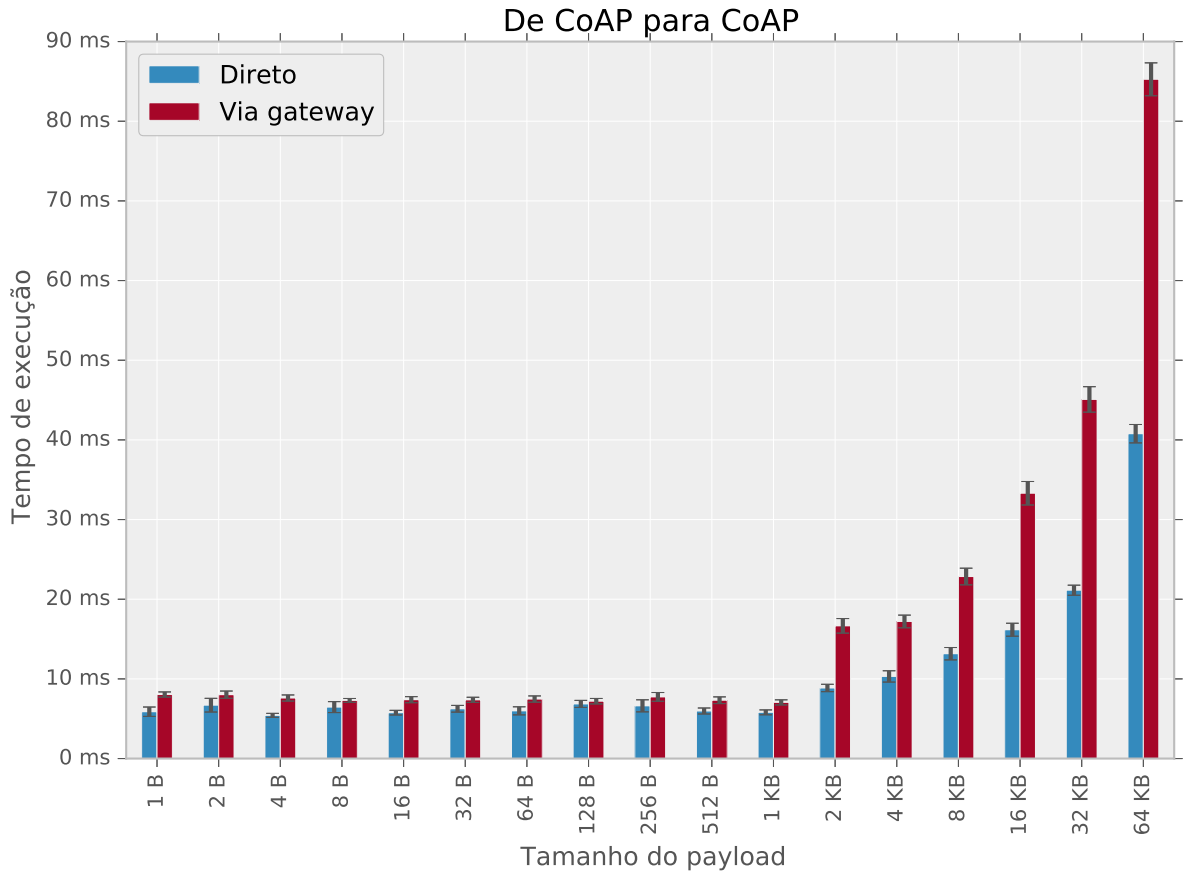


Figura 5.3 – Resultados do tempo de execução para requisições CoAP-CoAP.

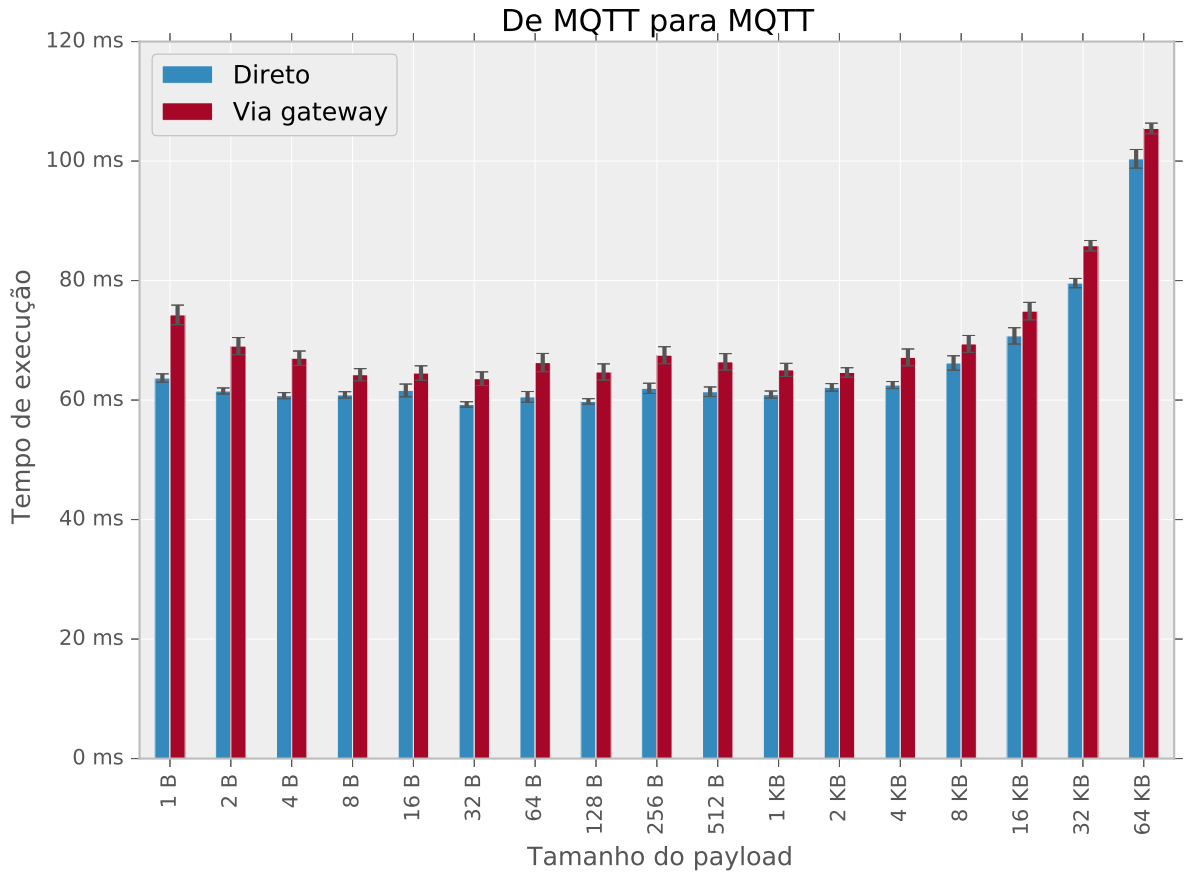


Figura 5.4 – Resultados do tempo de execução para requisições MQTT-MQTT.

Tabela 5.1 – Impacto da comunicação via *gateway* para requisições HTTP-HTTP.

Payload	Direto	Via *gateway*	Tempo extra	Impacto na comunicação
1 B	12,71 ms	20,60 ms	7,89 ms	62% mais lenta
2 B	12,45 ms	20,28 ms	7,83 ms	63% mais lenta
4 B	12,51 ms	19,96 ms	7,45 ms	60% mais lenta
8 B	12,83 ms	19,42 ms	6,59 ms	51% mais lenta
16 B	12,58 ms	19,26 ms	6,68 ms	53% mais lenta
32 B	12,62 ms	19,26 ms	6,64 ms	53% mais lenta
64 B	12,73 ms	19,22 ms	6,49 ms	51% mais lenta
128 B	12,73 ms	19,07 ms	6,34 ms	50% mais lenta
256 B	12,83 ms	19,13 ms	6,30 ms	49% mais lenta
512 B	12,90 ms	19,47 ms	6,57 ms	51% mais lenta
1 KB	12,88 ms	19,40 ms	6,52 ms	51% mais lenta
2 KB	12,78 ms	19,51 ms	6,73 ms	53% mais lenta
4 KB	12,95 ms	19,63 ms	6,68 ms	52% mais lenta
8 KB	12,79 ms	20,69 ms	7,90 ms	62% mais lenta
16 KB	14,01 ms	22,34 ms	8,33 ms	59% mais lenta
32 KB	19,12 ms	29,85 ms	10,73 ms	56% mais lenta
64 KB	32,28 ms	46,92 ms	14,64 ms	45% mais lenta

Tabela 5.2 – Impacto da comunicação via *gateway* para requisições CoAP-CoAP.

Payload	Direto	Via *gateway*	Tempo extra	Impacto na comunicação
1 B	5,88 ms	8,04 ms	2,16 ms	37% mais lenta
2 B	6,70 ms	8,02 ms	1,32 ms	20% mais lenta
4 B	5,42 ms	7,60 ms	2,18 ms	40% mais lenta
8 B	6,46 ms	7,30 ms	0,84 ms	13% mais lenta
16 B	5,76 ms	7,39 ms	1,63 ms	28% mais lenta
32 B	6,25 ms	7,37 ms	1,12 ms	18% mais lenta
64 B	5,99 ms	7,47 ms	1,48 ms	25% mais lenta
128 B	6,86 ms	7,20 ms	0,34 ms	5% mais lenta
256 B	6,62 ms	7,74 ms	1,12 ms	17% mais lenta
512 B	5,97 ms	7,32 ms	1,35 ms	23% mais lenta
1 KB	5,80 ms	7,05 ms	1,25 ms	22% mais lenta
2 KB	8,86 ms	16,66 ms	7,80 ms	88% mais lenta
4 KB	10,31 ms	17,21 ms	6,90 ms	67% mais lenta
8 KB	13,16 ms	22,85 ms	9,69 ms	74% mais lenta
16 KB	16,17 ms	33,30 ms	17,13 ms	106% mais lenta
32 KB	21,13 ms	45,08 ms	23,95 ms	113% mais lenta
64 KB	40,78 ms	85,26 ms	44,48 ms	109% mais lenta

Tabela 5.3 – Impacto da comunicação via *gateway* para requisições MQTT-MQTT.

Payload	Direto	Via *gateway*	Tempo extra	Impacto na comunicação
1 B	63,68 ms	74,25 ms	10,57 ms	17% mais lenta
2 B	61,52 ms	69,03 ms	7,52 ms	12% mais lenta
4 B	60,74 ms	67,00 ms	6,26 ms	10% mais lenta
8 B	60,87 ms	64,24 ms	3,37 ms	6% mais lenta
16 B	61,61 ms	64,52 ms	2,91 ms	5% mais lenta
32 B	59,26 ms	63,58 ms	4,32 ms	7% mais lenta
64 B	60,54 ms	66,27 ms	5,74 ms	9% mais lenta
128 B	59,77 ms	64,70 ms	4,93 ms	8% mais lenta
256 B	61,96 ms	67,51 ms	5,55 ms	9% mais lenta
512 B	61,39 ms	66,40 ms	5,01 ms	8% mais lenta
1 KB	60,92 ms	65,05 ms	4,13 ms	7% mais lenta
2 KB	62,12 ms	64,62 ms	2,50 ms	4% mais lenta
4 KB	62,51 ms	67,14 ms	4,63 ms	7% mais lenta
8 KB	66,20 ms	69,39 ms	3,19 ms	5% mais lenta
16 KB	70,74 ms	74,88 ms	4,14 ms	6% mais lenta
32 KB	79,58 ms	85,83 ms	6,25 ms	8% mais lenta
64 KB	100,37 ms	105,46 ms	5,09 ms	5% mais lenta

6 Conclusões e Trabalhos Futuros

Foi exposta nesta dissertação a questão da heterogeneidade na Internet das Coisas (IoT), com foco nos protocolos da camada de aplicação, também chamados de protocolos de mensagem. Embora a adoção de um padrão por todos solucionaria o problema, isso não é possível na IoT, pois os dispositivos restritos não suportam todos os tipos de protocolo.

A solução proposta para esse problema é GoThings, uma arquitetura com diretrizes para criar *gateways* capazes de interconectar protocolos de mensagem da camada de aplicação. Quanto aos padrões de troca de mensagem, a arquitetura suporta tanto requisição-resposta quanto publicar-assinar. Os aspectos principais desta arquitetura são interoperabilidade, extensibilidade, generalidade, configurabilidade e adequação a dispositivos restritos.

A arquitetura GoThings habilita a interoperabilidade transparente de uma forma que as aplicações podem usar apenas seus próprios protocolos para estabelecer comunicações heterogêneas. Esta arquitetura foi projetada de forma extensível: habilitar um novo protocolo é só uma questão de instalar um novo *plugin*. E é genérica o suficiente no contexto de problemas da IoT, através de um modelo que habilita aos usuários poderem alterar os comportamentos de um *gateway*.

Além da definição da arquitetura, este trabalho contribuiu com uma implementação desta, onde foram mapeados e, consequentemente, interconectados os protocolos HTTP, CoAP e MQTT. Embora nem todos os recursos definidos na arquitetura puderam ser desenvolvidos em tempo, esse protótipo serviu para provar o conceito. O desempenho do *gateway* implementado, embora não tenha sido o foco deste trabalho, foi aceitável, não causando um impacto tão grande para a maioria das aplicações da IoT.

6.1 Trabalhos Futuros

Como trabalhos futuros, podemos destacar os seguintes:

- Melhorar a implementação, removendo as limitações observadas na Seção 4.7.3.4.
- Especificar um formato de serialização leve, que permita desacoplar a linguagem de programação utilizada nos *plugins*.
- Estender o modelo de mensagem para habilitar interações REST, tomando o cuidado para continuar compatível com os padrões de troca de mensagem.
- Estender a arquitetura para permitir escalabilidade horizontal, isto é, poder adicionar mais *gateways*, distribuindo as tarefas.
- Requisições onde o cliente e o servidor são do mesmo protocolo atualmente passam pelo mesmo processo de mapeamento das requisições heterogêneas, aumentando o *overhead*.

Assim, é necessário que o *gateway* seja capaz de pular as etapas de mapeamento quando a requisição for homogênea sem perder recursos importantes do *gateway* como o cache.

Referências

APACHE HttpComponents. 2015. Disponível em: <http://hc.apache.org/>. Acesso em: 8 de out. de 2015.

BANKS, A.; GUPTA, R. (Ed.). *MQTT Version 3.1.1*. 2014. Disponível em: <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html>. Acesso em: 2 de nov. de 2014.

BERGMANN, O. *libcoap: C-Implementation of CoAP*. 2016. Disponível em: <https://libcoap.net/>. Acesso em: 15 de mar. de 2016.

BERNERS-LEE, T. et al. RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax. *Request for Comments*, Internet Engineering Task Force, aug 1998. Disponível em: <http://tools.ietf.org/html/rfc2396>.

BLAIR, G. S. et al. Interoperability in Complex Distributed Systems. In: BERNARDO, M.; ISSARNY, V. (Ed.). *Formal Methods for Eternal Networked Software Systems*. Springer Berlin Heidelberg, 2011. v. 6659, p. 1–26. ISBN 978-3-642-21454-7. Disponível em: <http://dx.doi.org/10.1007/978-3-642-21455-4_1>.

BOTTERMAN, M. Internet of Things: an early reality of the Future Internet. In: *Workshop Report, European Commission Information Society and Media*. Prague: [s.n.], 2009.

BROMBERG, Y.-D. et al. Automatic Generation of Network Protocol Gateways. In: BACON, J. M.; COOPER, B. F. (Ed.). *ACM/IFIP/USENIX, 10th International Middleware Conference*. Urbana, Illinois: Springer Berlin Heidelberg, 2009. (Lecture Notes in Computer Science, v. 5896), p. 21–41. ISBN 978-3-642-10444-2. Disponível em: <http://dx.doi.org/10.1007/978-3-642-10445-9_2>.

BUSCHMANN, F.; HENNEY, K.; SCHMIDT, D. C. *Pattern Oriented Software Architecture: A Pattern Language for Distributed Computing*. West Sussex: John Wiley Sons, 2007. ISBN 978-0-470-05902-9.

CALIFORNIUM (Cf) CoAP framework. 2016. Disponível em: <https://www.eclipse.org/californium/>. Acesso em: 17 de jan. de 2016.

CASTELLANI, A.; FOSSATI, T.; LORETO, S. HTTP-CoAP cross protocol proxy: an implementation viewpoint. In: *IEEE 9th International Conference on Mobile Ad-Hoc and Sensor Systems (MASS 2012)*. Las Vegas, NV: IEEE, 2012. Supplement, p. 1–6. ISBN 978-1-4673-2433-5. Disponível em: <http://dx.doi.org/10.1109/MASS.2012.6708523>.

CHAPPELL, D. A. *Enterprise Service Bus: Theory in Practice*. EUA: O'Reilly Media, 2004. (Theory in practice). ISBN 978-0-596-00675-4.

COLLINA, M.; CORAZZA, G. E.; VANELLI-CORALLI, A. Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST. In: *IEEE International Symposium on Personal, Indoor and Mobile Radio Communications, PIMRC*. Sydney, NSW: IEEE, 2012. p. 36–41. ISBN 9781467325691. ISSN 2166-9570. Disponível em: <http://dx.doi.org/10.1109/PIMRC.2012.6362813>.

CONTIKI: The Open Source OS for the Internet of Things. 2014. Disponível em: <http://contiki-os.org/>. Acesso em: 10 de dez. de 2014.

COULOURIS, G. et al. *Sistemas Distribuídos: Conceitos e Projeto*. 5. ed. Porto Alegre: Bookman, 2013. ISBN 978-85-8260-053-5.

CURL and libcurl. 2016. Disponível em: <https://curl.haxx.se/>. Acesso em: 2 de fev. de 2016.

DARGIE, W.; POELLABAUER, C. *Fundamentals of wireless sensor networks: theory and practice*. West Sussex, United Kingdom: John Wiley Sons Ltd., 2010. ISBN 978-0-470-99765-9.

ELMANGOUSH, A. et al. Application-derived communication protocol selection in M2M platforms for smart cities. In: *2015 18th International Conference on Intelligence in Next Generation Networks*. IEEE, 2015. p. 76–82. ISBN 978-1-4799-1866-9. Disponível em: <http://dx.doi.org/10.1109/ICIN.2015.7073810>.

EVANS, D. *How the Next Evolution of the Internet Is Changing Everything*. 2011. Disponível em: <http://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf>. Acesso em: 2 de nov. de 2014.

FIELDING, R.; RESCHKE, J. RFC 7231 - Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. *Request for Comments*, Internet Engineering Task Force, jun 2014. ISSN 2070-1721. Disponível em: <http://tools.ietf.org/html/rfc7231>.

FIELDING, R. T.; TAYLOR, R. N. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology*, ACM, v. 2, n. 2, p. 115–150, 2002. ISSN 15335399. Disponível em: <http://dx.doi.org/10.1145/514183.514185>.

FLURRY, G.; CLARK, K. J. *The Enterprise Service Bus, re-examined*. 2011. Disponível em: <http://www.ibm.com/developerworks/websphere/techjournal/1105_flurry/1105_flurry.html>. Acesso em: 8 de out. de 2015.

FOSTER, A. *Messaging Technologies for the Industrial Internet and the Internet of Things Whitepaper*. 2014. Disponível em: <http://www.prismtech.com/download-documents/1561>. Acesso em: 2 de nov. de 2014.

FREED, N.; BORENSTEIN, N. S. RFC 2046 - Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. *Request for Comments*, 1996. Disponível em: <http://tools.ietf.org/html/rfc2046>.

GUDGIN, M. et al. (Ed.). *SOAP Version 1.2*. 2007. Disponível em: <https://www.w3.org/TR/soap12/>. Acesso em: 11 de dez. de 2015.

GUINARD, D.; TRIFA, V.; WILDE, E. A resource oriented architecture for the Web of Things. In: *Internet of Things (IOT)*. Tokyo: IEEE, 2010. p. 1–8. ISBN 978-1-4244-7413-4. Disponível em: <http://dx.doi.org/10.1109/IOT.2010.5678452>.

HARTKE, K. RFC 7641 - Observing Resources in the Constrained Application Protocol (CoAP). *Request for Comments*, Internet Engineering Task Force, 2015. Disponível em: <https://tools.ietf.org/html/rfc7641>.

HUNKELER, U.; TRUONG, H. L.; STANFORD-CLARK, A. MQTT-S — A publish/subscribe protocol for Wireless Sensor Networks. In: *3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE '08)*. IEEE, 2008. p. 791–798. ISBN 978-1-4244-1796-4. Disponível em: <http://dx.doi.org/10.1109/COMSWA.2008.4554519>.

ISSARNY, V.; BENNACEUR, A.; BROMBERG, Y.-D. Middleware-Layer Connector Synthesis: Beyond State of the Art in Middleware Interoperability. In: BERNARDO, M.; ISSARNY, V. (Ed.). *Formal Methods for Eternal Networked Software Systems*. Springer Berlin Heidelberg, 2011, (Lecture Notes in Computer Science, v. 6659). p. 217–255. ISBN 978-3-642-21454-7. Disponível em: <http://dx.doi.org/10.1007/978-3-642-21455-4_7>.

KARAGIANNIS, V. et al. A Survey on Application Layer Protocols for the Internet of Things. *Transaction on IoT and Cloud Computing*, ICAS, v. 3, n. 1, p. 11–17, 2015. ISSN 2331-4761. Disponível em: <http://icas-pub.org/ojs/index.php/ticc/article/view/47>.

LORETO, S. et al. RFC 6202 - Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP. *Request for Comments*, Internet Engineering Task Force, apr 2011. Disponível em: <https://tools.ietf.org/html/rfc6202>.

MACÊDO, W. L. d. A. M.; ROCHA, T. da; MORENO, E. D. GoThings - An Application-layer Gateway Architecture for the Internet of Things. In: *Proceedings of the 11th International Conference on Web Information Systems and Technologies, WEBIST*. Lisbon, Portugal: SCITEPRESS, 2015. p. 135–140. ISBN 978-989-758-106-9. Disponível em: <http://dx.doi.org/10.5220/0005493701350140>.

MOQUETTE MQTT broker. 2015. Disponível em: <https://github.com/andsel/moquette>. Acesso em: 18 de out. de 2015.

NANOHTTPD. 2015. Disponível em: <http://nanohttpd.org/>. Acesso em: 2 de set. de 2015.

NETTY. 2015. Disponível em: <http://netty.io/>. Acesso em: 27 de jul. de 2015.

PAHO - Open Source messaging for M2M. 2015. Disponível em: <https://eclipse.org/paho/>. Acesso em: 21 de ago. de 2015.

PRIESTLEY, T. *The Internet Of Things Is A Fragmented $ 19 Trillion Roulette Gamble*. 2015. Disponível em: <http://www.forbes.com/sites/theopriestley/2015/10/05/the-internet-of-things-is-a-fragmented-19-trillion-roulette-gamble/>. Acesso em: 26 de out. de 2015.

RIVEST, R. RFC 1321 - The MD5 Message-Digest Algorithm. *Request for Comments*, Internet Engineering Task Force, apr 1992. Disponível em: <https://tools.ietf.org/html/rfc1321>.

RODRÍGUEZ-DOMÍNGUEZ, C. et al. A communication model to integrate the Request-Response and the Publish-Subscribe paradigms into ubiquitous systems. *Sensors (Basel, Switzerland)*, Molecular Diversity Preservation International, v. 12, n. 6, p. 7648–7668, jan 2012. ISSN 1424-8220. Disponível em: <http://dx.doi.org/10.3390/s120607648>.

SHELBY, Z.; HARTKE, K.; BORMANN, C. RFC 7252 - The Constrained Application Protocol (CoAP). *Request for Comments*, Internet Engineering Task Force, 2014. ISSN 2070-1721. Disponível em: <https://tools.ietf.org/html/rfc7252>.

SHENG, Z. et al. A survey on the IETF protocol suite for the Internet of Things: standards, challenges, and opportunities. *IEEE Wireless Communications*, v. 20, n. 6, p. 91–98, dec 2013. Disponível em: <http://dx.doi.org/10.1109/MWC.2013.6704479>.

TANENBAUM, A. S.; WETHERALL, D. J. *Computer Networks*. 5. ed. Boston: Pearson Prentice Hall, 2011. ISBN 978-0-13-212695-3.

TIME(1). 2015. Disponível em: <http://man7.org/linux/man-pages/man1/time.1.html>. Acesso em: 26 de jan. de 2016.